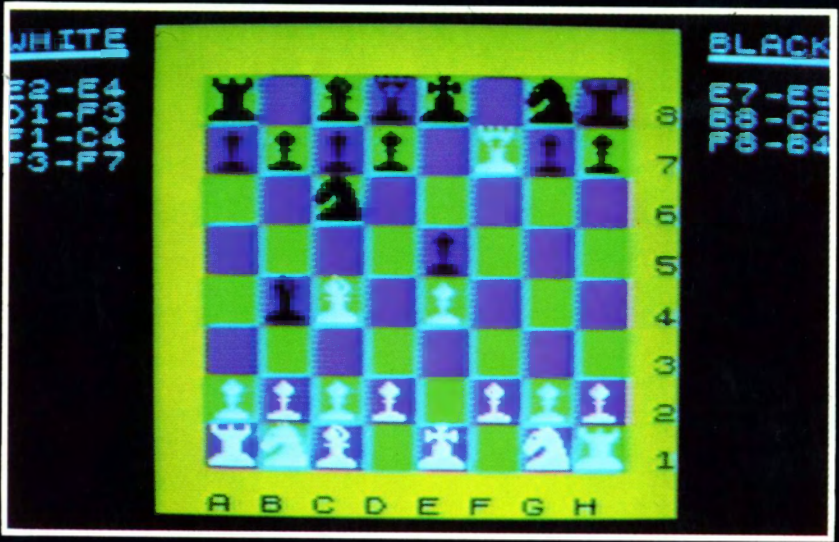
Advanced Graphics with the Sinclair ZX Spectrum



Advanced Graphics with the Sinclair ZX Spectrum

Advanced Graphics with the Sinclair ZX Spectrum

Ian O. Angell and Brian J. Jones

Department of Statistics and Computer Science,
Royal Holloway College,
University of London,
Egham, Surrey

MACMILLAN PRESS
LONDON

First published 1983
Reprinted (with corrections) 1983

Published by
THE MACMILLAN PRESS LTD
London and Basingstoke
Companies and representatives
throughout the world

Printed in Great Britain at
The Camelot Press Ltd,
Southampton

ISBN 0 333 35050 2 (book)
35051 0 (cassette)

Contents

Preface

With the rapid advance of computer technology has come a substantial reduction in the price of computer hardware. In the coming years the price of peripheral devices will also tumble. This means that users with a limited budget, who previously had access only to the most elementary computing devices, will soon be able to afford the most sophisticated computers. They will also be able to escape from the limitation of tabular numerical output and buy microprocessor attachments for television monitors or inexpensive special-purpose colour graphics devices. Sinclair computers have always led the field in this respect. Software, however, does not appear to be getting cheaper.

Because of the enormous capital expenditure required to set up graphical output in the past, both in machines and software, the subject of computer graphics has been the preserve of large research groups. This inaccessibility has led to a mystique growing up around the subject and it has achieved a false reputation for difficulty. This book is an attempt to lay the ghost of complexity; it will also show that complicated (and hence expensive) software packages, which are naturally of great value in research organisations, need not frighten away the average computer user. For most purposes these packages are unnecessary. This book, as well as being an introduction to computer graphics, may be considered a (very inexpensive) software package: it is a lot cheaper than commercially available packages! Naturally, because of this fundamental approach, users have to achieve a reasonable understanding of their graphics device before pictures, other than those provided, may be drawn. This need not be a disadvantage; the amount of groundwork required will be seen to be very limited. As a direct result, the knowledge of the user grows along with the package and he is far less likely to misinterpret any of the graphical routines. References are given and relevant further reading material is recommended in order to expand the reader's horizons in the subject.

It is assumed that the reader has an elementary knowledge of Cartesian coordinate geometry (the authors recommend the books detailed in Cohn (1961), Coxeter (1974), and McCrae (1953) – see References), and also of the BASIC programming language (see the Spectrum BASIC Handbook (Vickers (1982) and Hurley (1982)). Many interesting programming exercises are proposed, and these should raise the standard of the reader's BASIC expertise. BASIC is a universally popular language, available (in various guises) on all types of microcomputer, so the programs may be easily adjusted to run on machines other than the Spectrum: it is also a good medium for transmitting the algorithms used in computer graphics, enabling readers to translate these ideas readily into any other computer language of their choice.

The concepts necessary for the study of computer graphics are organised as a combination of theory and worked examples; these are introduced as and when needed in the natural progression of the subject. Some program listings form part of the examples and these should not be considered just as algorithms that describe solutions to fundamental graphical problems, but also as a computer graphics software package in BASIC, or simply as programs to draw patterns. Alongside the examples are a series of exercises that expand these ideas. The practical problems implicit in programming the various concepts of computer graphics are often more a source of difficulty to the student than the concepts themselves. Therefore it is essential that readers implement many of the program listings given in the book in order to understand the algorithms, as well as attempt a large number of exercises. As an extra learning aid, a companion audio-cassette tape is being made available; this contains most of the larger program listings given in this book. If readers are nervous of the mathematics, they should run the programs first before studying the theory.

This approach to the subject has been used with great success in teaching computer graphics to undergraduates and postgraduates at Royal Holloway College. Quickly producing apparently complex pictures results in the positive feedback of enthusiastic interest. The ability to construct pictures on line-drawing and colour interactive graphics screens makes a long-lasting impression on students; and the step-by-step approach brings them very quickly to the level of very sophisticated computer graphics. That level is outside the scope of this book, but where necessary readers will find relevant references to guide them into the more advanced topics.

This book is aimed at those who are competent BASIC programmers but complete beginners in graphics. It contains the elementary ideas and basic information about pixel and two-dimensional graphics, which must be mastered before attempting the more involved ideas of character and three-dimensional graphics. This is followed by a section relating to character graphics and the display of data (in line drawings and colour) - probably the most important non-specialised, commercial use of computer graphics. Later chapters introduce the reader to the geometry of three-dimensional space, and to a variety of projections of this space on to the two-dimensional space of graphics devices. The related problems of hidden lines and hidden surfaces, as well as the construction of complex three-dimensional objects, are dealt with in detail. Finally, we return to advanced ideas in BASIC programming and a large worked example of a video game.

Graphics is one of the most rapidly expanding areas of computer science. It is being used more and more in the fields of Computer Aided Design (CAD), Computer Assisted Management (CAM) and Computer Assisted Learning (CAL). At one time it was only the big corporations such as aircraft and automobile manufacturers that used these techniques, but now most companies are realising the potential and financial savings of these ideas. What is more, not only is computer graphics profitable, it's fun! The Sinclair Spectrum is an ideal machine on which to learn the basics of computer graphics, and an excellent springboard to the most sophisticated (and expensive) graphics devices.

We hope this book will display some of the excitement and enthusiasm for computer graphics experienced by us, our colleagues and students. To demonstrate just how useful computer drawings are for illustrating books and pamphlets, all the pictures here were drawn by computer specifically for this book.

Introduction

This book may be read at a number of different levels. Firstly, it can be considered as a recipe book of graphics programs for those who simply want to draw complex pictures with their Spectrum. We naturally hope that the reader, having drawn these figures, will be inspired to delve deeper into the book in order to understand how and why the programs were constructed. Secondly, some of the programs may be used as a package to produce and label data diagrams (pie-charts, histograms and graphs) for business and laboratory use. Finally, and the main reason for writing the book, it is an introductory text to computer graphics, which leads the reader from the elementary notions of the subject through to such advanced topics as character graphics, construction of three-dimensional objects and hidden line (and surface) algorithms.

The complex programs later in the book are much too involved to be given as a single listing. Furthermore we will see a great deal of repetition in the use of elementary algorithms. Therefore we use the *top down* or *modular* approach in writing and explaining programs. The solution to each major graphics problem is conceived as a series of solutions to subproblems. These subproblems may be further broken down into a set of problems to be solved (*modules*). These modules will be programmed in the form of BASIC subroutines. Each is given an identifier (in lower case characters) and will solve a particular subtask. Then the totality of submodules combine to solve the required graphics problem. Because the program listings are used to represent algorithms for the solution of these subtasks, we decided in general not to use statements like GO SUB 6000. We prefer instead to assign the subroutine identifier to the address value at the beginning of the routine (for example, LET scene3 = 6000) and thus we can write statements like GO SUB scene3. We use lower case for subroutine identifiers (and groupings of routines in the text) only: all other program variables will be in upper case to avoid confusion.

Spectrum BASIC does not have the facility of passing parameters into routines. Values of input parameters have to be set in assignment statements outside the routine, and the names of output parameters must be known if sensible use is to be made of the routine. This can be rather inconvenient if you are using someone else's package of routines. It is essential that users know the names of the input and output parameters; therefore in our routines we use the REMarks IN: (to identify the INput parameters) and OUT: (for the OUTput parameters). We number our programs so that all program statements are on lines ending in 0, and REMarks on lines ending in 1 to 9 (except the naming of routines). The IN: and OUT: REMarks follow directly the naming REMark on lines ending in 1 and 2 respectively. Also the cassette tape listings of programs use character codes to highlight and colour various REMarks (see chapter 13). In cases where we think that the word REM detracts from readability of a line we use these codes to make it invisible. We have minimised the REMarks on the cassette so that we can pack the maximum amount of program listing on to the tape. It is a good idea to expand these listings by adding the complete REMarks, and SAVE them on your own tapes.

For those who want only to run our programs, we give a list of complete programs at the end of each chapter together with suitable data values. In fact it is a good idea for all, including the serious readers, to SAVE the routines on tape before approaching each chapter. They can then LOAD, MERGE and RUN the programs as they occur in the text. The cassette tape available to accompany the text contains all the larger listings in the book, as well as BYTE data for diagrams and character sets used in later programs (which would otherwise have to be constructed by readers themselves, a rather time-consuming process). Our routines were written for the 48K Spectrum: if you have a 16K machine you should read appendix A and note the changes that need to be made.

As an example of what to expect, we give below the program required to draw figure I.1, a line drawing of a body of revolution in which all the hidden lines have been suppressed. This will work on both types of machine.

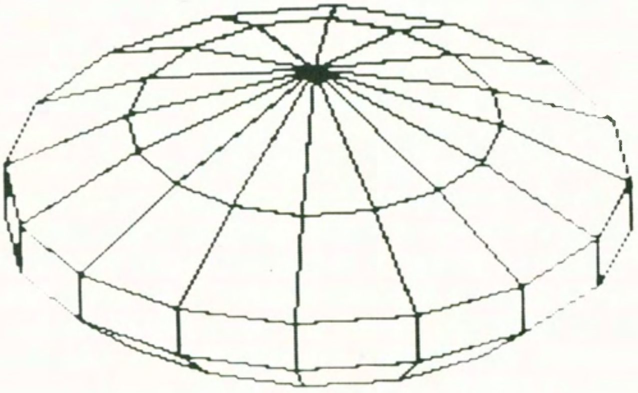


Figure I.1

The program requires the MERG(E)ing of listings 2.1 ('start'), 2.2 (two functions FN X and FN Y), 2.3 ('setorigin'), 2.4 ('moveto') and 3.3 ('clip' and 'lineto'). This combination of routines will be called 'lib1', and it was designed for drawing line figures on the television screen.

To 'lib1' must be added listings 3.4 ('angle'), 8.1 ('mult3' and 'idR3'), 8.2 ('tran3'), 8.3 ('scale3'), 8.4 ('rot3'), 9.1 ('look3') and 9.2 ('main program').

Routines, which when combined we call 'lib3', are used for transforming and observing objects in three-dimensional space.

We need also listing 10.3 ('revbod') as well as the 'scene3' routine given in listing I.1 below.

Listing I.1

```
6000 REM scene3/flying saucer
6010 DIM X(12): DIM Y(12)
6020 DIM S(6): DIM T(6)
6030 DIM A(4,4): DIM B(4,4): DIM R(4,4)
6040 DATA 0,3, 3,2, 5,1, 5,0, 4,-1, 0,-3
6050 RESTORE scene3
6060 LET revbod = 6500
6069 REM create object.
6070 LET NUMV = 5
6080 INPUT "NUMBER OF HORIZONTAL LINES",NUMH
6090 INPUT "ANGLE PHI ";PHI
6100 FOR I = 1 TO NUMV + 1: READ S(I),T(I): NEXT I
6109 REM position the observer.
6110 GO SUB idR3: GO SUB look3
6129 REM draw object.
6120 GO SUB revbod
6130 RETURN
```

Figure I.1 requires the data HORIZ = 12, VERT = 8, EX = 1, EY = 2, EZ = 3, DX = 0, DY = 0, DZ = 0, NUMH = 16 and PHI = 0. Each value has to be typed in individually on request by the machine. The picture will take about 5 minutes to draw, so be patient. Run the program with different data values. What happens if HORIZ = 6 and VERT = 4, and the other values stay the same? Set HORIZ = 15, VERT = 10, EX = 1, EY = –2, EZ = 3, DX = 1, DY = 0 and DZ = 0. Try NUMH = 20, PHI = 0.1. You will have to read up to and including chapter 10 to understand the details of what is happening.

This example illustrates the reasoning behind the layout of this book. Assuming that you are a fast typist, or that you have bought the accompanying tape, then a relatively complex three-dimensional picture can be constructed very quickly with a minimum of fuss. Even one-finger typists (like the authors) will have little difficulty in implementing this and the other programs, before they go on to study the book in detail.

We hope that this example will inspire you to implement all the programs in this book, to try most of the examples, and then to go on to draw your very own computer graphics pictures.

Now you can read the rest of our book and we wish you many happy hours with your Spectrum.

1 Graphics Operations of the ZX Spectrum

Throughout the course of this book we will be assuming that the reader is reasonably familiar with the BASIC programming language on the ZX Spectrum. In this chapter, however, we shall be looking at some of the BASIC commands – those concerned wholly or partly with graphics. With a series of example programs and simple exercises we shall examine and explore the Spectrum's capabilities. In the chapters that follow we shall use this knowledge to develop a sound understanding, both practical and mathematical, of computer graphics.

Initially we shall consider the hardware and software facilities available for producing pictures. All microcomputers that produce television pictures generate their graphical display using RASTER SCAN technology. This is also true of most of the newer commercial mini and main-frame computers. An area of memory is reserved to hold the display information for the screen and this is examined, bit by bit, as the electron beam sweeps across the raster screen. The display is composed of points, each of which is represented by a single bit (a binary on/off switch) in the memory. In the simplest case the beam is switched on for a short period each time a binary on is found, thus producing a point of light on the screen.

PAPER and INK

On the Spectrum we are given two commands; these directly control the way that the points are displayed. This affects the picture, which is made up of INK dots (binary ons) on a PAPER background (binary offs). The commands, named PAPER and INK (naturally), are called by using the name followed by a number N $(0 \leqslant N \leqslant 9)$.

PAPER N   sets the background colour of the picture. After this statement is executed, all newly generated binary offs in the memory will be displayed in colour N (that is, until another PAPER command is executed).

INK N   sets the points of light corresponding to binary ons to colour N in a similar way.

The number N, when in the range 0 to 7, represents the colour printed above the corresponding numeric key on the keyboard. If N is 8, then the colour previously set for an area is used. If N is 9, then the colour of PAPER/INK is set to either black or white and will contrast with the other INK/PAPER colour currently in use. In general, black INK on white PAPER is clearest, as is obvious from any book, and this is the normal setting the for Spectrum.

Display File

This type of picture is referred to as a *memory-mapped* display since it corresponds directly to the contents of an area of memory. On the Spectrum this part of the memory is known as the *display file* and starts at location 16384. A simple exploration of how the display is affected by changing the contents of the memory can be made with a program such as listing 1.1.

Listing 1.1

```
10 LET CORNER = 16384
20 LET VALUE = 137
30 POKE CORNER,VALUE
40 STOP
```

This program uses POKE to store a VALUE (entered as a decimal) in the first location of the display file. This location holds the information for the top left-hand CORNER of the screen. Since each location, or *byte*, contains eight binary bits, the first eight points on the display are affected. These change to show a pattern equivalent to the binary representation of the VALUE: in this case 10001001.

Exercise 1.1

(i) Experiment with different VALUEs and change the program either to,
 (a) use BIN (binary) representation for VALUE, or to
 (b) use a FOR. . .NEXT loop to change VALUE.

(ii) Use the PAPER and INK commands to change the background and foreground colours and then re-run the program to see what difference this makes.

BORDER

When the PAPER colour is changed it soon becomes obvious that we cannot write on the whole of the screen. A BORDER is left around the edge of the PAPER to avoid the distortion at the edge of the screen suffered by all television displays. The colour of this BORDER can be changed, in a similar way to the PAPER and INK colours, by the command

BORDER N

where N is in the range 0 to 7 and indicates the new BORDER colour.

Character Blocks

A complete picture can be built up by storing various VALUEs at locations in the display memory in a similar way to listing 1.1. For instance, we could store the eight VALUEs 0, 98, 148, 136, 136, 136, 148, 98 in the display-file locations that represent the start of eight consecutive lines on the screen (see listing 1.2). We see the pattern of INK dots corresponding to the 'ones' shown in figure 1.1.

	128	64	32	16	8	4	2	1	
00000000 =									= 0
01100010 =		64 +	32				+ 2		= 98
10010100 =	128			+ 16		+ 4			= 148
10001000 =	128				+ 8				= 136
10001000 =	128				+ 8				= 136
10001000 =	128				+ 8				= 136
10010100 =	128			+ 16		+ 4			= 148
01100010 =		64 +	32				+ 2		= 98

Figure 1.1

This is the way in which characters are defined (and redefined) on the Spectrum, but we shall leave further investigation of this until chapter 5. Nevertheless it does illustrate that a picture, even as small as this, takes time to prepare and requires a comparatively complicated program to produce the display.

Listing 1.2

```
10 LET CORNER = 16384
20 LET LINE = 256
30 DATA 0,98,148,136,136,136,148,98
40 FOR I = 0 TO 7
50 LET MEMORY = CORNER + I*LINE
60 READ VALUE
70 POKE MEMORY,VALUE
80 NEXT I
90 STOP
```

PLOT and DRAW

We have seen how the screen display can be changed by storing different values in the display file. But there are over six thousand locations in the display file and changing each of these individually would be quite tedious. We obviously need a more effective method of changing the display.

BASIC provides us with graphics commands to deal with this problem, the simplest of which are PLOT and DRAW. All the graphics commands treat the display as a grid of 256 points horizontally by 176 points vertically (45056 in total). These points are known as *pixels* and are individually identified by a pair of integers. The graphics commands help in constructing pictures by allowing us to control a *graphics pen*, which is initially positioned over pixel (0, 0). We can now explain these commands.

PLOT X, Y moves our pen to pixel (X, Y) and plots an INK point there.

DRAW X,Y draws a line from our pen's current position to the point, X pixels away horizontally and Y pixels away vertically. If X is negative, the point will be to the left and if X is positive, it will be to the right. Similarly if Y is negative, the point will be below our old position, or if Y is positive, above.

After the execution of these commands, the pen remains over the last pixel to be visited, awaiting the next command. Before examining the other more advanced graphics commands, we shall first see what is possible using only lines and/or points.

We are now in a position to draw large-scale pictures on the screen. For instance, we can draw a box around that area of screen available for graphics (listing 1.3).

Listing 1.3

```
 10 PLOT 0,175
 20 PLOT 255,175
 30 PLOT 255,0
 40 PLOT 0,0
 50 IF INKEY$ <>"" THEN GO TO 50
 60 IF INKEY$ = "" THEN GO TO 60
 70 DRAW 0,175
 80 DRAW  255,0
 90 DRAW 0,-175
100 DRAW -255,0
110 STOP
```

This program first PLOTs points at the corners of the PAPER; it then waits until a key is pressed before joining them up by DRAWing lines around the boundary of the PAPER. On comparing the PLOT and DRAW commands we see that there is an important difference in the way they work: the PLOT command uses the *absolute* pixel coordinates, whereas the DRAW command uses the *relative* positions of the points. This means that, in order to draw a line segment between two pixel points on the screen, it is first necessary to use PLOT to move the graphics pen to the point at one end of a line segment, then work out the position of the second end point relative to the first, before finally the line may be DRAWn. Note that in listing 1.3 all the points are decided before the program is run. In general, points are more likely to be INPUT, READ or calculated while the program is running.

Exercise 1.2
Write a program that calculates the position of lines to draw a grid. DRAW them using two FOR. . .NEXT loops (one for horizontal lines, the other for vertical lines).

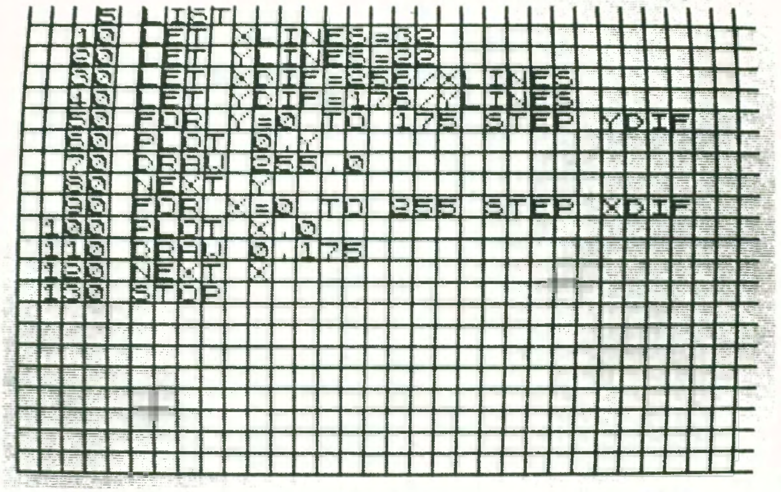
Exercise 1.3
Write a program that accepts N pairs of pixel coordinates as INPUT from the keyboard, and then DRAWs an irregular polygon of N sides by joining the points in order. (This requires some careful thought since the first point must be joined to the last.)

PRINT and LIST

So far we have not discussed the most obvious method of changing the display, namely using the PRINT and LIST commands. This is because these commands use character-size blocks and are designed primarily for use with low-resolution graphics. This topic will be dealt with in chapter 5 but, since the Spectrum allows high-resolution and low-resolution graphics to be freely intermixed, we give a small example here. Suppose we add the line

```
5 LIST
```

to the start of the program for exercise 1.2, and then set the program to draw a grid of 32 vertical lines and 22 horizontal lines. We get a display similar to figure 1.2, which shows the size and position of the character blocks.



```
   5 LIST
  10 LET XLINES=32
  20 LET YLINES=22
  30 LET XDIF=255/XLINES
  40 LET YDIF=175/YLINES
  50 FOR Y=0 TO 175 STEP YDIF
  60 PLOT 0,Y
  70 DRAW 255,0
  80 NEXT Y
  90 FOR X=0 TO 255 STEP XDIF
 100 PLOT X,0
 110 DRAW 0,175
 120 NEXT X
 130 STOP
```

Figure 1.2

We can use the PLOT command to demonstrate the high-resolution capabilities of the Spectrum by drawing *fractals* (see Mandelbrot, 1977).

To draw a simple fractal we follow this routine. Imagine a square with sides of length 4^n. This may be divided into 16 smaller squares, each with sides of length 4^{n-1}, which we number 1 to 16 as in figure 1.3. Four of these smaller squares, numbers 2, 8, 9 and 15, are rearranged to produce figure 1.4.

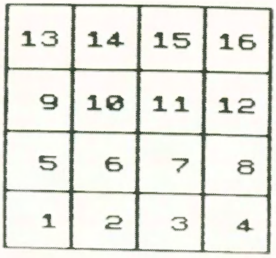


Figure 1.3

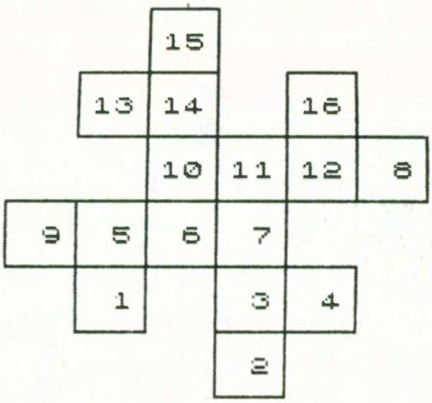


Figure 1.4

Each of the squares in the pattern is now split up into 16 even smaller squares, in the same way, and these are similarly rearranged. We repeat this process until we have squares with sides of length 1. The resulting fractal pattern consists entirely of unit squares, which we can PLOT as single pixels. The program in listing 1.4 starts from a square with sides of length 64, which is 4^3; thus in the program there must be three FOR. . .NEXT loops nested inside each other. The final picture produced is shown in figure 1.5.

Listing 1.4

```
 10 DIM X(16): DIM Y(16)
 20 FOR I = 1 TO 4
 30 FOR J = 1 TO 4
 40 LET K = 4*I + J - 4
 50 LET X(K) = J - 3: LET Y(K)=I - 3
 60 NEXT J: NEXT I
 70 LET X(2)  =  0: LET Y(2)  = -3
 80 LET X(8)  =  2: LET Y(8)  =  0
 90 LET X(9)  = -3: LET Y(9)  = -1
100 LET X(15) = -1: LET Y(15) =  2
110 FOR I = 1 TO 16
120 FOR J = 1 TO 16
130 FOR K = 1 TO 16
140 LET XX = 16*X(I) + 4*X(J) + X(K)
150 LET YY = 16*Y(I) + 4*Y(J) + Y(K)
160 PLOT 128+XX,88+YY
170 NEXT K: NEXT J: NEXT I
180 STOP
```

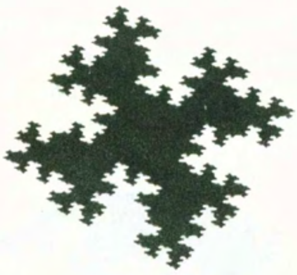


Figure 1.5

INVERSE and OVER

We shall now consider the options that affect the way in which lines and points are placed on the screen. There are two commands, and to use them we must enter the command name followed by a number. The number is 1 to turn the effect on, and 0 to turn it off again.

INVERSE: while this effect is on, all lines or points will be draw in the background (PAPER) colour. That is, the binary switches will be turned off instead of on.

OVER: while this effect is on, any pixel affected by a graphics command will be flipped to its opposite state. Any pixel of INK is changed to the PAPER colour, and vice versa. That is, the binary switch for the pixel is flipped over to the other position.

Using these commands we can produce programs that generate seemingly complex patterns and rapidly changing displays. Listing 1.5 gives a program that combines two methods of creating complicated patterns from very simple instructions.

Listing 1.5

```
 10 OVER 1
 20 LET LINES = 400
 30 LET A = 0: LET ANGLE = 2*PI/LINES
 40 FOR I = 1 TO LINES
 50 LET X = 85*COS A
 60 LET Y = 85*SIN A
 70 PLOT 128,88
 80 DRAW X,Y
 90 LET A = A + ANGLE
100 NEXT I
110 OVER 0
120 STOP
```

On a display composed of discrete points (pixels), angled lines will be drawn as a series of short, horizontal or vertical steps. When two such lines are drawn close together at slightly different angles, many of the steps on the lines will overlap. Consider figure 1.6, drawn by listing 1.5. The lines that form the central area overlap each other many times and so this area would be a mass of black were OVER not used. With OVER on, those pixels that lie on an odd number of lines are switched on, whereas the others remain off. This produces the striking pattern at the centre of the figure. On the other hand the outer area of the pattern is produced by holes, left by the line steps, of pixels not lying on any line.

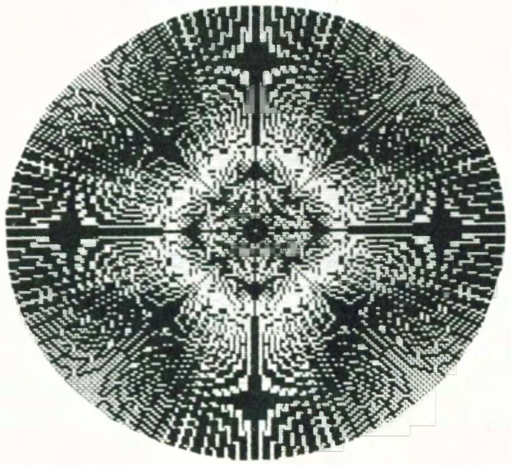


Figure 1.6

Exercise 1.4

Alter listing 1.5 to INPUT the value of LINES and also to INPUT a string variable, indicating whether or not the OVER option is to be used. Use this program to explore the parts played by OVER and by the steps on adjacent lines as LINES is varied.

The repeated use of random numbers to produce a variety of rapidly changing graphical displays has been a favourite device for attracting attention to computers. Listing 1.6 shows one simple illustration of this method using RND and OVER to place pixels at random about the screen.

Listing 1.6

```
10 OVER 1
20 LET X = INT (RND*256)
30 LET Y = INT (RND*176)
40 PLOT X,Y
50 BEEP 0.05,(X - Y)/10
60 GO TO 20
```

Exercise 1.5
Alter listing 1.6 to DRAW lines, either between the random points as they are generated, or from the centre of the screen (128, 88) to each point.

In the above exercise we saw that the OVER option ensured that the display changed with each command, even if the same command was repeated, for example, by DRAWing the same point, or line. The OVER option may be used in this way to display an object briefly – by drawing it twice, once to put it on the screen and again to take it off. Listing 1.7 moves a point around the screen by PLOTting it at its new position and immediately PLOTting its last position again to remove the old point.

Listing 1.7

```
 10 OVER 1: PAPER 0: INK 7: BORDER 4: CLS
 20 LET SPEED = 2
 30 LET X = 0: LET Y = 0
 40 LET XADD = SPEED: LET YADD =  SPEED
 50 PLOT X,Y
 60 LET OLDX = X: LET OLDY = Y
 70 LET X = X + XADD
 80 IF X > 255 - SPEED OR X < SPEED THEN LET XADD = -XADD
 90 LET Y = Y + YADD
100 IF Y > 174 - SPEED OR Y < SPEED THEN LET YADD = -YADD
110 PLOT X,Y
120 PLOT OLDX,OLDY
130 GO TO 60
```

We can extend this program to allow keyboard control of the moving point (listing 1.8). The lower case letters about "f" enable the point to move in eight separate directions under our control. If a "p" is typed then the point leaves a trailing line that shows its past movements: if a "q" is typed then the point ceases to leave a trail.

This type of animation is an important and commonly used technique. We shall use it extensively, both in programs like the game in chapter 14, and in programs like the 'cursor' routine in chapter 6.

Listing 1.8

```
 10 OVER 1
 20 LET X = 0: LET Y = 0
 30 PLOT X,Y
 40 LET OLDX = X: LET OLDY = Y
 50 LET XADD = 0: LET YADD = 0
 60 LET A$ = INKEY$: IF A$ = "" THEN GO TO 60
 70 IF A$ = "p" THEN OVER 0
 80 IF A$ = "q" THEN OVER 1
 90 IF (A$ = "e" OR A$ = "d" OR A$ = "c") AND X > 0 THEN LET XADD = -1
100 IF (A$ = "c" OR A$ = "v" OR A$ = "b") AND Y > 0 THEN LET YADD = -1
110 IF (A$ = "t" OR A$ = "g" OR A$ = "b") AND X < 255 THEN LET XADD = 1
120 IF (A$ = "e" OR A$ = "r" OR A$ = "t") AND Y < 175 THEN LET YADD = 1
130 IF XADD = 0 AND YADD = 0 THEN GO TO 60
140 LET X = X + XADD: LET Y = Y + YADD
150 PLOT X,Y
160 PLOT OLDX,OLDY
170 GO TO 40
```

We can achieve large-scale animation by using lines to extend or contract a polygonal area. Listing 1.9 uses this method together with the INVERSE command to form a fast zoom effect.

Listing 1.9

```
 10 LET I = 0: INK 9
 20 LET UP = 175: LET ACROSS = 255
 30 LET X = 0: LET Y = 0
 40 LET DIF = 1
 50 INVERSE I
 60 PLOT X,Y
 70 DRAW 0,UP: DRAW ACROSS,0
 80 PLOT X,Y
 90 DRAW ACROSS,0: DRAW 0,UP
100 LET X = X + DIF: LET Y = Y + DIF
110 LET UP = UP - 2 * DIF
120 LET ACROSS = ACROSS - 2 * DIF
130 IF UP < 0 OR UP = 175 THEN LET DIF = -DIF: LET I = 1 - I
140 IF UP = 175 THEN PAPER RND*7: CLS
150 GO TO 50
```

Exercise 1.6
Draw a solid square composed of 40 by 40 pixels. Move this area about the screen under keyboard control. Note that you need change only the edges of the square.

FLASH and BRIGHT

Having seen what is possible in black and white, we shall now turn our attention to colour. The Spectrum can have all the colours on the screen at once, but inside each character block there can be only two colours, PAPER and INK. These colours may be BRIGHT and/or FLASHing; also, special effects like OVER and INVERSE can be turned on or off in the same way.

FLASH: when FLASH is set for a given block, the colours within that block will alternate between INK on PAPER and PAPER on INK.

BRIGHT: blocks with BRIGHT set will show both PAPER and INK at increased BRIGHTness. This has the effect of making non-BRIGHT blocks look darker.

FLASH and BRIGHT can also be set to 8, so that the pre-existing setting for a block is unchanged by PRINT.

Attributes

The current combinations of FLASH, BRIGHT, PAPER and INK colours for each character block are stored in memory in the *attribute file*. This contains one location for each of the character blocks; it is located in memory immediately after the display file, and starts at location 22528. We can use listing 1.10, a modified version of listing 1.1, to alter these values directly, as we did with the display file.

Listing 1.10

```
10 LET CORNER = 22528
20 INPUT "VALUE = BIN "; LINE V$
30 LET VALUE = VAL ("BIN " + V$)
40 PRINT AT 0,0;"*"
50 POKE CORNER,VALUE
60 GO TO 20
```

Exercise 1.7
Use the program from listing 1.10 to alter individual bits within the VALUE stored in the first location of the attribute file.

This VALUE affects the whole of the first character block by indicating for points in that block whether FLASH and BRIGHT are on or off, and what PAPER and INK colours are used. These pieces of information make up a BINary number in the following manner

FLASH – 0 or 1; BRIGHT – 0 or 1; PAPER – 000 to 111; INK – 000 to 111

Thus we can calculate the meaning of a value in the attribute file in the way shown in figure 1.7.

		FLASH	BRIGHT	PAPER	INK
Normal settings	=	1	0	5	2
BINary equivalent	=	1	0	101	010
BIN 10101010	=	128 +	32 +	8 +	2 = 170

Figure 1.7

The above example is the attribute for FLASHing, non-BRIGHT, cyan PAPER and red INK. The attribute value for any character block can be found using the function ATTR (ROW, COLUMN). The ROW and COLUMN parameters specify the position of the block by counting the number of ROWs down from the top line and the number of COLUMNs across from the left edge. These are the same parameters we use to PRINT AT a character block. So it is easy to write a program that changes the attributes of blocks at random (see listing 1.11). This has the same effect as randomly POK(E)ing values into the attribute file (see page 117 of the Spectrum BASIC Handbook (Vickers, 1982)).

Listing 1.11

```
10 FLASH  INT (RND*2)
20 BRIGHT INT (RND*2)
30 PAPER  INT (RND*8)
40 INK    INT (RND*8)
50 LET ROW = INT (RND*22): LET COL =  INT (RND*32)
60 PRINT AT ROW,COL;"*"
70 BEEP 0.02,RND*40
80 GO TO 10
```

Exercise 1.8
Modify the above program so that it changes just one character block to a random attribute setting, and then calculates and displays the FLASH, BRIGHT, PAPER and INK settings from the ATTR function.

For convenience we can use table 1.1 to convert between attribute file value and the attribute settings.

Table 1.1 Attribute Conversion

PAPER	INK								MODE
	Black	Blue	Red	Magneta	Green	Yellow	Cyan	White	
Black	0	1	2	3	4	5	6	7	NORMAL
	64	65	66	67	68	69	70	71	BRIGHT
	128	129	130	131	132	133	134	135	FLASH
	192	193	194	195	196	197	198	199	BRIGHT + FLASH

Blue	8	9	10	11	12	13	14	15	NORMAL
	72	73	74	75	76	77	78	79	BRIGHT
	136	137	138	139	140	141	142	143	FLASH
	200	201	202	203	204	205	206	207	BRIGHT + FLASH
Red	16	17	18	19	20	21	22	23	NORMAL
	80	81	82	83	84	85	86	87	BRIGHT
	144	145	146	147	148	149	150	151	FLASH
	208	209	210	211	212	213	214	215	BRIGHT + FLASH
Magenta	24	25	26	27	28	29	30	31	NORMAL
	88	89	90	91	92	93	94	95	BRIGHT
	152	153	154	155	156	157	158	159	FLASH
	216	217	218	219	220	221	222	223	BRIGHT + FLASH
Green	32	33	34	35	36	37	38	39	NORMAL
	96	97	98	99	100	101	102	103	BRIGHT
	160	161	162	163	164	165	166	167	FLASH
	224	225	226	227	228	229	230	231	BRIGHT + FLASH
Cyan	40	41	42	43	44	45	46	47	NORMAL
	104	105	106	107	108	109	110	111	BRIGHT
	168	169	170	171	172	173	174	175	FLASH
	232	233	234	235	236	237	238	239	BRIGHT + FLASH
Yellow	49	49	50	51	52	53	54	55	NORMAL
	112	113	114	115	116	117	118	119	BRIGHT
	176	177	178	179	180	181	182	183	FLASH
	240	241	242	243	244	245	246	247	BRIGHT + FLASH
White	56	57	58	59	60	61	62	63	NORMAL
	120	121	122	123	124	125	126	127	BRIGHT
	184	185	186	187	188	189	190	191	FLASH
	248	249	250	251	252	253	254	255	BRIGHT + FLASH

We can think of each pixel on a colour television screen as three dots of light packed closely together at the vertices of an equilateral triangle. For each pixel there is one red, one blue and one green dot, and the attribute-file locations are used to control the illumination of the three different colours. The display file will indicate that a given pixel is to be plotted in a particular INK colour. The lowest three bits (bits 0 to 2) of the attribute value for the block containing that pixel are used to decide whether the green, red and blue dots of that pixel are on or off. Our eyes contain only three types of colour sensor (green, red and blue). Our brain takes the signals from the three dots and combines them into a single dot of composite colour. So if the last three bits of the attribute are 111, equivalent to colour 7, we get green, plus red, plus blue. This corresponds to white light or white INK. The other colour codes, when written in binary form, can be translated in this way (see figure 1.8).

Colour	Number	Binary	Illuminated Dots
Black	0	000	
Blue	1	001	Blue
Red	2	010	Red
Magenta	3	011	Red + Blue
Green	4	100	Green
Cyan	5	101	Green + Blue
Yellow	6	110	Green + Red
White	7	111	Green + Red + Blue

Figure 1.8

When a PAPER-coloured pixel is to be illuminated, the three bits of the attribute corresponding to the PAPER colour (bits 3 to 5) are decoded in exactly the same way. Bit 6 in the attribute indicates whether BRIGHT is on or not, and is used to control the brightness at which all the dots will be illuminated. When an attribute has the FLASH bit set (bit 7), the colours corresponding to INK and PAPER will be alternated. The speed of alternation (that is, the FLASHing) depends on an internal value in the computer: on the Spectrum it changes about every one-third of a second.

In general, we may safely use two colours for high-resolution graphics, but beware! No more than two colours may occur in one character block at any one time. Subject to this proviso, full-colour high-resolution graphics may be achieved.

Exercise 1.9

Experiment with different colours using the programs in this chapter. Certain colour combinations can be just too much for a normal television set to cope with. Unless you are using an expensive monitor instead of a television screen, a combination of clashing colours for the program in listing 1.5 should produce a rather interesting effect of waves washing across the screen.

Simple Animation

We can produce more animated effects in low resolution by using colours and FLASH. Listing 1.12 shows some interesting techniques of colour animation. The first part of the program is particularly useful because the display needs no maintenance once set up. The boundary of the picture is a sequence of blocks composed of alternative blocks of FLASHing red PAPER and cyan INK, and FLASHing cyan PAPER and red INK. On seeing this our brains are tricked into believing that the red and cyan colours are moving around the boundary sequence.

Listing 1.12

```
 10 FLASH 1: PAPER 2: INK 5
 20 FOR I = 0 TO 1
 30 FOR J = 0 TO 15
 40 PRINT AT 0,2*J + I;" "
 50 PRINT AT 21,2*J + 1 - I;" "
 60 NEXT J
 70 FOR J = 0 TO 10
 80 PRINT AT 2*J + I,0;" "
 90 PRINT AT 2*J + 1 - I,31;" "
100 NEXT J
110 PAPER 5: INK 2
120 NEXT I
130 FLASH 0: LET P = 0
140 FOR I = 1 TO 20
150 PRINT AT I,1; PAPER P;"
160 LET P = P + 1: IF P = 7 THEN LET P = 0
170 NEXT I
180 GO TO 140
```

Exercise 1.10
Write low-resolution colour versions of the bouncing point program and the other animation programs. In your programs move character blocks instead of pixels around the screen.

CIRCLE and DRAW

The Spectrum has two further built-in high-resolution graphics commands that we have not yet examined: the CIRCLE command and the DRAW command for curved lines.

CIRCLE X, Y, R draws a CIRCLE of radium R pixels centred on pixel (X, Y). It is important to remember that after obeying this command our graphics pen is left at a pixel on the right-hand side of the CIRCLE, just below the centre.

DRAW X, Y, A DRAWs a curved line from the current position to the relative position X, Y, while turning through an angle A. This curved line will be an arc of a CIRCLE. The angle through which the line turns is specified in radians and may be between –PI and PI.

Listing 1.13 demonstrates the use of these commands by producing the display shown in figure 1.9.

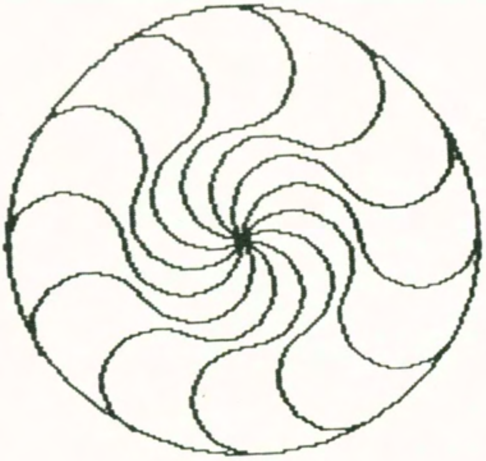


Figure 1.9

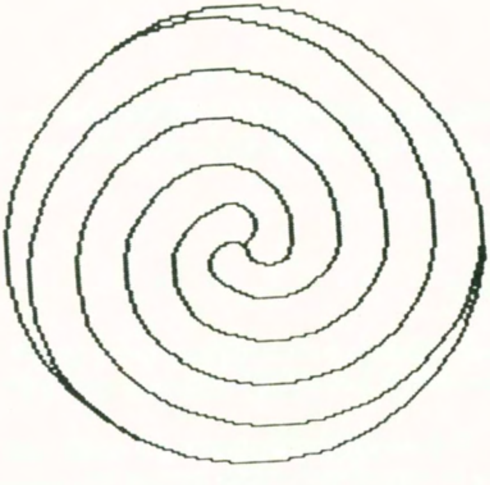


Figure 1.10

Listing 1.13

```
10 CIRCLE 128,88,80: LET N = 12
20 LET A = 0: LET ADIF = 2*PI/N
30 FOR I = 1 TO N
40 PLOT 128,88
50 LET X = 40*COS A : LET Y = 40*SIN A
60 DRAW X,Y,-PI: DRAW X,Y,PI
70 LET A =A + ADIF
80 NEXT I
```

Exercise 1.11

Figure 1.10 shows the traditional Celtic pattern known as a triskele. Write a program to draw this type of pattern.

Colours within a Character Block

The use of different INK colours for high-resolution graphics can cause problems and produce results that are calculable but usually unforeseen. Run the program in listing 1.14. It will show just how easily things can go wrong when more than two colours are used without careful planning.

Listing 1.14

```
10 PAPER 5: INK 7: CLS
20 FOR I = 0 TO 175 STEP 8
30 PLOT 0,I: DRAW 255,0
40 NEXT I
50 INK 0
60 CIRCLE 128,88,80
70 STOP
```

This problem can be used to our advantage. We can produce rapidly changing, and complicated, low-resolution colour displays. Initially we PRINT solid INK blocks at specified positions on the screen. Any line drawn subsequently will change the colour of the low-resolution blocks through which it passes. The impressive speed of this technique can be seen by running the program given in listing 1.15.

Listing 1.15

```
10 INK 7: CLS
20 FOR I = 1 TO 704: PRINT "█";: NEXT I
30 LET DIST = 80: LET I = 0: LET D = 8
40 INK I
50 PLOT 120,86 + DIST: DRAW DIST,-DIST: DRAW -DIST,-DIST
60 PLOT 127,86 - DIST: DRAW -DIST,DIST: DRAW DIST,DIST
70 LET DIST  = DIST - D: IF DIST = 0 OR DIST = 80 THEN LET D = - D
80 LET I = I + 1: IF I = 8 THEN LET I = 0
90 GO TO 40
```

A Simple Game

We now include a small game program (listing 1.16) as a final example of the use of the techniques discussed in this chapter. A worm can move in character block steps about the screen, horizontally or vertically, under control of the keyboard. The aim of the game is for the worm to eat the money (or target). The worm gets longer whenever it eats the target. If at any time the head of the worm runs headlong into the boundary, or into its own body, then the worm dies. After ten successful meals the worm returns to its original size, with a fanfare. The game then continues.

This game was developed using modular, structured methods preferred by programmers. These methods help to produce quickly a working and understandable program. Put simply, we must approach the program as a series of small tasks that build up block by block into the completed program. For this game these tasks were tentatively defined as

A. Initialise variables
B. Set up board
C. Control game
D. Update and print score

From this overview of the problems we can set about solving each problem or, if necessary, split them into yet smaller, more manageable problems. For example, task C above could be split into

1. Generate target
2. Use keyboard to change direction of worm
3. Move worm

Task 3 could be further split

a. Draw worm
b. Worm hits boundary or itself, and dies
c. Worm eats money and grows
d. Fanfare

No specific order is implied in this breakdown; for example, you may find that you want to regenerate the target from inside the fanfare section of program. These headings are simply a list of tasks that reflect the problems that come to mind when attempting the solution of a larger problem.

Examine the game below and try to identify which tasks are carried out, where, in what order, and which have been further subdivided. (Throughout this book the variable names in lower case will refer to line numbers at the start of subroutines. This helps to make the program more readable, gives a clear picture of the algorithm, and hence is good general practice.)

Note the use of logical expressions (for example, IF DEAD THEN. . .): see chapter 13 of the Spectrum BASIC Handbook (Vickers, 1982). Also note the use of ATTR and SCREEN$ to detect collisions, both by the colour of character blocks and by their contents. Figure 1.11 shows a typical state of the game.

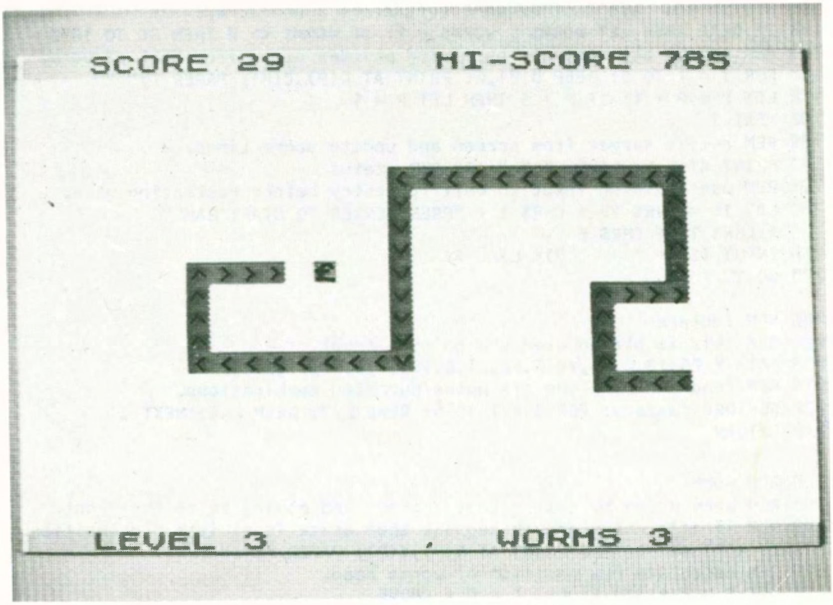


Figure 1.11

Listing 1.16

```
1000 DIM R(55): DIM C(55)
1009 REM initialise routine pointers and set hiscore to 0.
1010 LET fanfare = 2000: LET worm = 3000: LET key = 4000
     : LET gobble = 5000: LET status = 6000: LET  target = 7000
1020 BORDER 1: PAPER 7: INK 0
1030 LET HSC = 0
1039 REM start/restart for game.
1040 LET SCORE = 0: LET WORMS = 3: LET LEVEL = 1
1048 REM start a new worm, five segments long from row R column C.
1049 REM P is pointer to segment which is to be moved.
1050 LET S = 5: LET P = 1: LET R = 0: LET C = INT (RND*32)
1059 REM set movement variables so that worm is going down.
1060 LET RMOVE = 1: LET CMOVE = 0: LET H$ = "v"
1069 REM clear array of segment positions.
1070 FOR I = 1 TO 55: LET R(I) = -1: NEXT I
1079 REM set truth flags for game (0 = false, not 0 = true).
1080 LET WON = 0: LET DEAD = 0
1089 REM set up screen with yellow strips on top and bottom.
1090 CLS: PRINT AT 0,0; PAPER 6;,,: PRINT AT 21,0; PAPER 6;,,
1099 REM print out scores and place a £ note target on the screen.
1100 GO SUB status: GO SUB target
1109 REM main loop of game: check for controls,move worm.
```

```
1110 GO SUB key: GO SUB worm
1119 REM if nothing has happened keep on looping.
1120 IF NOT DEAD AND NOT WON THEN GO TO 1110
1129 REM if one worm has eaten £10 give fanfare and start a new worm.
1130 IF WON THEN LET LEVEL = LEVEL + 1: GO SUB fanfare: GO TO 1050
1139 REM make crashing noise.
1140 IF DEAD THEN FOR I = 1 TO 10: BEEP 0.005,15: NEXT I
1149 REM if you have another worm left start a new worm.
1150 IF DEAD THEN LET WORMS = WORMS - 1: IF WORMS <> 0 THEN GO TO 1050
1159 REM remove all segments by moving pointer along from back of worm.
1160 FOR I = 1 TO S: BEEP 0.01,0: PRINT AT R(P),C(P); PAPER 7;" "
1170 LET P = P + 1: IF P > S THEN LET P = 1
1180 NEXT I
1189 REM remove target from screen and update score lines.
1190 PRINT AT Y,X; PAPER 7;" ": GO SUB status
1199 REM use flashing input to wait for entry before restarting game.
1200 LET I$ = CHR$ 18 + CHR$ 1 + "PRESS ENTER TO START GAME"
     + CHR$ 18 + CHR$ 0
1210 INPUT (I$ + "      "); LINE A$
1220 GO TO 1040

2000 REM fanfare
2009 REM this is played when you go up a level.
2010 DATA 0.06,18,0.06,19,0.06,21,0.15,27,0.06,21,0.2,27
2019 REM read and play the six notes/duration combinations.
2020 RESTORE fanfare: FOR I = 1 TO 6: READ L,T: BEEP L,T: NEXT I
2030 RETURN

3000 REM worm
3008 REM worm moves by taking last segment and moving it to the front.
3009 REM if it's not a growth segment then erase it at it's old position.
3010 IF R(P) <> -1 THEN PRINT AT R(P),C(P); PAPER 7;" "
3019 REM calculate new position of worms head.
3020 LET R = R + RMOVE: LET C = C + CMOVE
3029 REM check for collision with boundaries.
3030 IF R > 20 OR R < 1 OR C > 31 OR C <0 THEN LET DEAD = 1: RETURN
3039 REM check for collision with another segment of worm.
3040 IF ATTR (R,C) = 16 THEN LET DEAD = 1: RETURN
3049 REM set row and column of segment to new position.
3050 LET R(P) = R: LET C(P) = C
3059 REM check whether target has been eaten.
3060 IF SCREEN$ (R,C) = "£" THEN GO SUB gobble
3069 REM put new segment on screen and move pointer along worms back.
3070 PRINT AT R(P),C(P); PAPER 2;H$
3080 LET P = P + 1: IF P > S THEN LET P = 1
3090 RETURN

4000 REM key
4009 REM check for controls being used.
4010 LET A$ = INKEY$: IF A$ = "" THEN RETURN
4019 REM make sure all letters are treated as lower case.
4020 IF CODE A$ < 96 THEN LET A$ = CHR$ (CODE A$ + 32)
4028 REM worm can only turn left or right from course not back on itself.
4029 REM control for up is pressed, if you're not going down then turn up.
4030 IF A$ = "i" AND CMOVE THEN LET RMOVE = -1: LET CMOVE = 0
     : LET H$ = "↑ ": RETURN
4039 REM turn down ( if worm is not going up ).
4040 IF A$ = "m" AND CMOVE THEN LET RMOVE = 1: LET CMOVE = 0
     : LET H$ = "v": RETURN
4049 REM turn left ( if worm is not going right ).
4050 IF A$ = "j" AND RMOVE THEN LET RMOVE = 0: LET CMOVE = -1
     : LET H$ = "<": RETURN
4059 REM turn right ( if worm is not going left ).
```

```
4060 IF A$ = "k" AND RMOVE THEN LET RMOVE = 0: LET CMOVE = 1
     : LET H$ = ">": RETURN
4070 RETURN

5000 REM gobble
5009 REM eat target, make gobbling noises.
5010 FOR I = 2 TO 4 STEP 0.5
5020 BEEP 0.01,EXP I - 10: NEXT I
5029 REM add to your score and update score-lines.
5030 LET SCORE = SCORE + 1: GO SUB status
5038 REM make five more segments available for growth.
5039 REM if worm has 55 segments then it has eaten £10 so you win a round.
5040 LET S = S + 5: IF S = 55 THEN LET WON = 1: RETURN
5049 REM place a new target on the screen.
5050 GO SUB target
5060 RETURN

6000 REM status
6009 REM if your score beats hiscore then update hiscore.
6010 IF SCORE > HSC THEN LET HSC = SCORE
6019 REM print out both score-lines.
6020 PRINT AT 0,0; PAPER 6;"   SCORE ";SCORE," HI-SCORE ";HSC
6030 PRINT AT 21,0; PAPER 6;"   LEVEL ";LEVEL,"   WORMS ";WORMS
6040 RETURN

7000 REM target
7009 REM choose a character block in the playing area at random.
7010 LET X = RND*31: LET Y = RND*19 + 1
7019 REM check that it's not the same place as the last one.
7020 IF X = C AND Y = R THEN GO TO target
7029 REM check that it's not under the worm.
7030 IF ATTR (Y,X) = 16 THEN GO TO target
7039 REM print new £ note on the screen.
7040 PRINT AT Y,X; PAPER 4;"£"
7050 RETURN
```

Exercise 1.12
As a final mini-project for this chapter, write a squash game or ping-pong video game (or both!) using low-resolution colour graphics. The ball can be a pixel or character block, and the bat(s) should be controlled from the keyboard like the worm in the above program. You will find it useful to turn some of the program sections from this chapter into subroutines.

In this chapter we have restricted ourselves to using the screen as a fixed piece of PAPER for patterns and games. To step up from pixel graphics to drawing pictures of real objects, we need commands that will relate the real world to our PAPER. In the following chapters we shall explore and develop the techniques needed to draw these real graphics pictures.

Complete Programs

I. Listing 1.1: no data required.
II. Listing 1.2: no data required.

III. Listing 1.3: no data required.
IV. Listing 1.4: no data required.
V. Listing 1.5: no data required.
VI. Listing 1.6: no data required.
VII. Listing 1.7: no data required.
VIII. Listing 1.8: press keys around character "F" to move point in one of eight directions, press "P" to start leaving a trail and "Q" to stop trail.
IX. Listing 1.9: no data required.
X. Listing 1.10: no data required.
XI. Listing 1.11: no data required.
XII. Listing 1.12: no data required.
XIII. Listing 1.13: no data required.
XIV. Listing 1.14: no data required.
XV. Listing 1.15: no data required.
XVI. Listing 1.16 (main program and routines 'fanfare', 'worm', 'key', 'gobble', 'status' and 'target'): use keys "I", "J", "K" and "M" to control movement of worm.

2 From Real Coordinates to Pixels

We have seen that Spectrum imagines its *graphics frame* to be a rectangular matrix of *addressable points* or *pixels*. These pixels are stacked in NXPIX (= 256) vertical columns and NYPIX (= 176) horizontal rows. Individuals from the set of NXPIX by NYPIX pixels can be uniquely identified by a bracketed pair of integers; these are sometimes called a pixel vector (I, J), where $0 \leqslant I \leqslant$ NXPIX $- 1$ and $0 \leqslant J \leqslant$ NYPIX $- 1$, the vector specifying the position of the pixel in the I^{th} column and J^{th} row: the vector (0, 0) identifies the bottom left-hand corner pixel of the frame. The Spectrum has its own set of BASIC instructions that enable users to operate on the matrix of pixels, treating them as points of light that can be switched off or on. This enables the operator to approximate lines, or polygons and other special types of area, with a series of coloured dots (the pixels).

The chapters that follow can be considered as taking the reader some way towards generating a two-dimensional and three-dimensional graphics package for the Sinclair Spectrum: the programs are given in BASIC and rely (with a few exceptions) on a small number of *primitive* routines given in this chapter.

Primitives that Map Continuous Space on to the Graphics Frame

In general, computer graphics deals with points, lines, areas and volumes in continuous two-dimensional and three-dimensional Euclidean space. Pixel graphics is very limited. The definition of objects that use only discrete pairs of integers is very rare in most practical applications. We therefore need to consider ways of plotting views of objects on a graphics screen, where positions are measured in real units: inches, miles or even light-years! Therefore we consider the relationship between two-dimensional real space and screen pixels. Before we can attempt this step, however, we must first discuss ways of representing two-dimensional space using Cartesian coordinate geometry.

We can imagine two-dimensional space as the plane of this page extending to infinity in all directions. Our description of the coordinate geometry starts by arbitrarily choosing a fixed point in this space, which we call the *coordinate origin*. Through the origin we draw a line that stretches to infinity in both directions; this is the *x-axis*. The normal convention is to place this line left to right on the page (the horizontal). Another two-way infinite line, the *y-axis*, is drawn through the origin perpendicular to the x-axis; conventionally this is placed from the top to the bottom of the page (the vertical). We now draw a scale along each axis: unit distances need not be the same on both axes, but this is normally the case (see figure 2.1). We assume that values on the x-axis are positive to the right of the origin and negative to the left: values on the y-axis are positive above the origin and negative below.

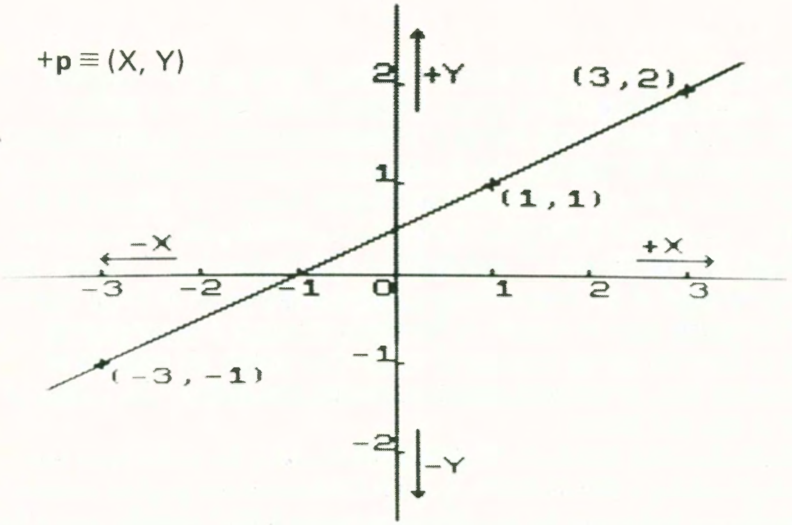


Figure 2.1

Taking any point p in space we can now uniquely fix its position by specifying its *coordinates* (figure 2.1). The *x-coordinate*, X say, is that distance along the x-axis (positive to the right of the axis and negative to the left) at which a line perpendicular to the x-axis passing through p cuts the x-axis. The *y-coordinate*, Y say, is correspondingly defined using the y-axis. These two values, called a *coordinate pair* or *two-dimensional vector*, are normally written in brackets thus: (X, Y). Note that the x-coordinate comes before the y-coordinate. We shall usually refer to the pair as a vector – the dimension (in this case two) will be understood from the context in which we use the term. A vector, as well as defining a point (X, Y) in two-dimensional space, can also be used to specify a direction, namely the direction that is parallel to the line joining the origin to the point (X, Y) – but more of this (and other objects such as lines, curves and polygonal areas) in chapter 3.

We are now in a position to devise means (the above-mentioned primitive routines) for mapping such geometrical concepts on to the two-dimensional discrete rectangular matrix of pixels that form the graphics frame.

Here we concentrate on two-dimensional space: an extension into three-dimensional space is dealt with starting at chapter 7. In both cases we require a method of mapping a rectangular area of two-dimensional Cartesian space on to the graphics frame. For simplicity we start by insisting that this area has its edges parallel to the *x*-axis and *y*-axis of Cartesian space. Initially, we assume that this rectangular area of space has its *bottom left-hand corner* identified with the co-ordinate origin (0.0, 0.0), while the length of the horizontal edge is HORIZ and the vertical edge VERT. We first identify the origin with the (0, 0) pixel of the frame, and then scale the rectangular area so that it fits into the frame; naturally the area only exactly fits the frame if the ratios HORIZ:VERT and NXPIX: NYPIX are equal (that is, 256:176). This is rarely the case, so we choose a scaling factor, XYSCALE, that maps the point (HORIZ, VERT) on to a pixel either on the upper or the right-hand edge of the frame. We can consider this rectangle as a *window* on to Cartesian space; no longer anchored to the coordinate origin, it may wander about space viewing rectangular areas of the same size as the original, but still having edges parallel to the original coordinate axes. As a general rule we make HORIZ roughly one-and-a-half times VERT.

At any time during the execution of the program we can move the coordinate origin from its original position at the bottom left-hand corner of the frame. Its position relative to the first origin will be stored as XORIG and YORIG, the *x*-component and *y*-component respectively. Initially, (XORIG, YORIG) is identified with (0.0, 0.0). Hence any point in Cartesian space with coordinates (XPT, YPT), a pair of reals, maps into a pixel with horizontal component INT((XORIG + XPT)*XYSCALE + 0.5) and vertical component INT((YORIG + YPT)* XYSCALE + 0.5). (INT is the BASIC function that truncates the fractional part of a decimal number and returns an integer.) These two components are stored as functions FN X and FN Y (see listing 2.2). During the construction of a picture we must consider a *plot pen*, in value a pair of integers, which moves about the graphics frame; initially it is placed at (0, 0), and in general it is the (XPEN, YPEN) pixel. The constants NXPIX and NYPIX, and the variables XYSCALE, XPEN, YPEN, XORIG and YORIG, must be available at all times to the plotting routines that follow, so these names must not be used for any other purpose. The routines were written specifically for the Spectrum, but we discuss also the general principles of constructing similar routines for other graphical devices. To start with we would have to change the values of NXPIX and NYPIX before we could use a different machine.

Our first routine 'start' initialises the required variables and prepares the screen for plotting. Listing 2.1 is an example 'start' routine for the Spectrum.

Listing 2.1

```
9700 REM start
9701 REM IN  : HORIZ, VERT
9702 REM OUT : NXPIX, NYPIX, XORIG, YORIG, XYSCALE, XPEN, YPEN
9710 LET XORIG = 0: LET YORIG = 0
```

```
9720 LET XPEN = 0: LET YPEN = 0
9730 LET NXPIX = 256: LET NYPIX = 176
9740 LET XYSCALE = NXPIX/HORIZ: LET YSCALE = NYPIX/VERT
9750 IF XYSCALE > YSCALE THEN LET XYSCALE = YSCALE
9760 RETURN
```

This routine may be extended should we need colour; it must have two extra parameters COLPAP and COLINK (integers between 0 and 7) for the colour of the paper and ink respectively. The following extra statement should be added

9725 PAPER COLPAP : INK COLINK

If we need to write this routine for a different microcomputer, all that is necessary is to replace the statements containing the Spectrum BASIC graphics instructions with the equivalent routines for the new machine. Most of the other routines in this book are independent of the structure of the Spectrum.

In 'start' and in many other routines that follow, it is necessary to transform the *x*/*y*-coordinates of a point into their pixel equivalents, so we introduce the two functions FN X and FN Y in listing 2.2.

Listing 2.2

```
9650 DEF FN X(Z) = INT ((XORIG + Z)*XYSCALE + 0.5)
9660 DEF FN Y(Z) = INT ((YORIG + Z)*XYSCALE + 0.5)
```

The next primitive routine (listing 2.3) is 'setorigin'. This enables us to move the coordinate origin by an amount XMOVE horizontally and YMOVE vertically (distances in the scale of the coordinate system), consequently adjusting the (XORIG, YORIG) values. After such a move the plot pen moves to the pixel equivalent of the new origin.

Listing 2.3

```
9600 REM setorigin
9601 REM IN  : XORIG, YORIG, XMOVE, YMOVE
9602 REM OUT : XORIG, YORIG, XPEN, YPEN
9610 LET XORIG = XORIG + XMOVE: LET YORIG = YORIG + YMOVE
9620 LET XPEN = FN X(0)
9630 LET YPEN = FN Y(0)
9640 RETURN
```

We shall be in a position to draw straight lines after we have produced two further routines: 'moveto', which moves the plot pen to a pixel equivalent of the point in coordinate space at one end of the line, and 'lineto', which draws the line by moving the plot pen from its present position (set by a previous call to 'setorigin', 'moveto' or 'lineto') to the pixel equivalent of the point on the other end of the line. Listings 2.4 and 2.5 show 'moveto' and 'lineto' routines designed specifically for the Spectrum. The 'lineto' routine includes statements that initiate the machine-dependent BASIC pixel instructions for drawing a line (note that PLOT is absolute and DRAW is relative); however, the 'moveto' routine is machine-independent. Hence if you wish to implement these routines on a different microcomputer you need only alter the 'lineto' routine.

Listing 2.4

```
9500 REM moveto
9501 REM IN  : XPT, YPT
9502 REM OUT : XPEN, YPEN
9510 LET XPEN = FN X(XPT)
9520 LET YPEN = FN Y(YPT)
9530 RETURN
```

Listing 2.5

```
9400 REM lineto
9401 REM IN  : XPT, YPT, XPEN, YPEN
9402 REM OUT : XPEN, YPEN
9410 LET NXPEN = FN X(XPT)
9420 LET NYPEN = FN Y(YPT)
9430 PLOT XPEN,YPEN
9440 DRAW NXPEN-XPEN,NYPEN-YPEN
9450 LET XPEN = NXPEN: LET YPEN = NYPEN
9460 RETURN
```

In all but the most elementary machines, it is possible to set up these plotting routines or their equivalents (and many more as our knowledge increases) in a library file or backing store. Then there is no need to retype them explicitly into each new program. On the Spectrum we can store them as files on audio-cassettes (and MERGE them if necessary). On the companion cassette to this book you will find these routines as part of the 'lib1' library.

Example 2.1
Identify a rectangle in Cartesian space, 30 units by 20 units, with the graphics frame of the Spectrum. Then draw a square of side 15 units, centred in the rectangle (figure 2.2a).

We centre the square by moving the origin to (15.0, 10.0) and thus define the corners of the square to be (±7.5, ±7.5). See listing 2.6.

Listing 2.6

```
100 REM drawing a square
109 REM setup identifiers to graphics routines.
110 LET start = 9700: LET setorigin = 9600: LET moveto = 9500
    : LET lineto = 9400
119 REM define graphics area.
120 LET HORIZ = 30: LET VERT = 20
130 GO SUB start
140 LET XMOVE = HORIZ*0.5: LET YMOVE = VERT*0.5
150 GO SUB setorigin
159 REM join corners of square in order.
160 LET XPT = 7.5: LET YPT = 7.5: GO SUB moveto
```

```
170 LET XPT = -7.5: LET YPT = 7.5: GO SUB lineto
180 LET XPT = -7.5: LET YPT = -7.5: GO SUB lineto
190 LET XPT = 7.5: LET YPT = -7.5: GO SUB lineto
200 LET XPT = 7.5: LET YPT = 7.5: GO SUB lineto
210 STOP
```

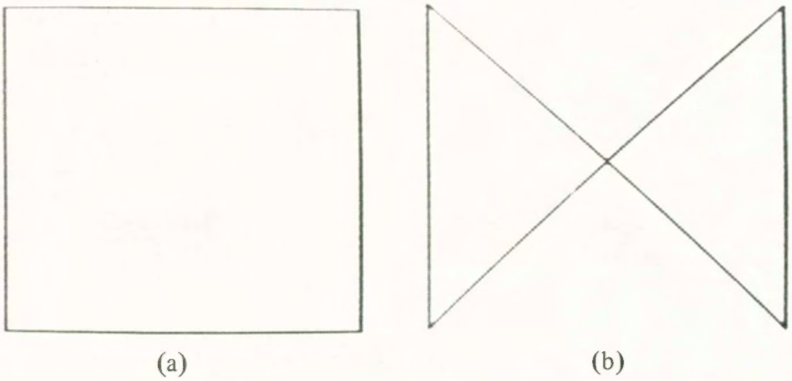

(a) (b)

Figure 2.2

It is as well to note, at this juncture, that the order in which the points are joined is critical. For example, if the coordinates of the second and third corners of the square are interchanged then figure 2.2b will be drawn.

Next we write a primitive routine 'polygon' that uses the Spectrum line-drawing instruction to draw such figures. The routine is given the NPOL vertices of the polygon as arrays X and Y (the *x*-coordinate and *y*-coordinate). We also give an example main program calling this routine (listing 2.7).

Listing 2.7

```
100 REM main program/ calling polygon
110 LET start = 9700: LET setorigin = 9600: LET moveto = 9500
    : LET lineto = 9400: LET polygon = 300
120 LET HORIZ = 30: LET VERT = 20
130 GO SUB start
140 LET XMOVE = HORIZ*0.5: LET YMOVE = VERT*0.5
150 GO SUB setorigin
159 REM declare and input vertices of polygon.
160 DIM X(50): DIM Y(50)
170 INPUT "TYPE NUMBER OF VERTICES  ",NPOL
180 FOR I = 1 TO NPOL
190 INPUT ("X("+ STR$ I+") ");X(I),("Y("+ STR$ I+") ");Y(I)
200 NEXT I
210 GO SUB polygon
220 STOP
```

```
300 REM polygon
301 REM IN  : NPOL, X(), Y()
310 LET XPT = X(NPOL): LET YPT = Y(NPOL): GO SUB moveto
319 REM join vertices of polygon in order.
320 FOR I = 1 TO NPOL
330 LET XPT = X(I): LET YPT = Y(I): GO SUB lineto
340 NEXT I
350 RETURN
```

Exercise 2.1
If we are using the Spectrum then it is possible to draw pictures in a variety of colours. But before drawing it is necessary to set the colour using the INK operation. Write a routine 'setcolour' with one integer parameter COLINK that achieves this.

Exercise 2.2
In all the plotting routines above, the scale of the mapping (XYSCALE) is fixed once and for all; and the horizontal and vertical scaling factors are identical. There is no need to heed this convention: write a routine 'factor' that alters the horizontal scale by FX and the vertical by FY. Naturally, this implies that we now have to define two separate scales (XSCALE and YSCALE, say); and also, of course, the 'start', 'setorigin', 'moveto' and 'lineto' routines must be altered (see also chapter 6).

Exercise 2.3
There is no reason for the x-axis and y-axis to be identified with the horizontal and vertical respectively. In fact they need not even be mutually perpendicular. Experiment with these ideas, which necessarily involves changing all the plotting routines 'start', 'moveto', etc.

Example 2.2
One of the first popular graphics packages was CalComp. This includes a number of routines to draw axes and scales for the construction of graphs, and many other useful subroutines. They are all based on a line-drawing routine named 'plot' (not to be confused with the Spectrum PLOT), which is central to the package; 'plot' has three parameters, two reals XPT and YPT, the coordinates of a point in space, and the movement information MOVE, an integer whose value is set to ±2 or ±3. This one routine may be used to replace all three of our routines 'setorigin', 'moveto' and 'lineto'. If MOVE is negative, then a new coordinate origin is fixed at the point (XPT, YPT) of the old coordinate system – equivalent to 'setorigin'. When the absolute value of MOVE is 3, then the plot head is moved without drawing a line – equivalent to 'moveto': when it is 2, then a line is drawn – equivalent to 'lineto'.

Naturally, even if we do not wish to implement the complete CalComp package, we can still implement the 'plot' routine in place of 'setorigin', 'moveto' and 'lineto'; and use it instead, in conjunction with the remaining routines mentioned in this chapter – see listing 2.8.

Listing 2.8

```
9800 REM plot / CalComp
9801 REM IN  : XPT, YPT, XPEN, YPEN, XORIG, YORIG, MODE
9802 REM OUT : XPEN, YPEN, XORIG, YORIG
9810 LET NXPEN = FN X(XPT)
9820 LET NYPEN = FN Y(YPT)
9830 IF ABS (MODE) = 2 THEN PLOT XPEN,YPEN: DRAW NXPEN-XPEN,NYPEN-YPEN
9840 LET XPEN = NXPEN: LET YPEN = NYPEN
9850 IF MODE < 0 THEN LET XORIG = XORIG + XPT: LET YORIG = YORIG + YPT
9860 RETURN
```

To demonstrate the use of these plotting routines we shall draw some simple patterns. There are those who think that the construction of patterns is a frivolous waste of time. Nevertheless, we consider it a very useful first stage in understanding the techniques of computer graphics. Often, patterns of an apparently sophisticated design are the result of very simple programs. Quickly producing such graphical output is an immediate boost to morale, and gives a lot of confidence to the beginner. Furthermore, new designs are always in demand: geometrical art is used for the covers of books and pamphlets and in advertising literature. It can do no harm to initiate artistic ideas that will be of great use later when we study the pictorial display of data. Patterns are also an ideal way of introducing some of the basic concepts of computer graphics in a very palatable way. Take the next example, which looks at the important role of trigonometric functions (sine and cosine), and of angular measurement in radians! Remember that π radians is the same angular measure as 180 degrees.

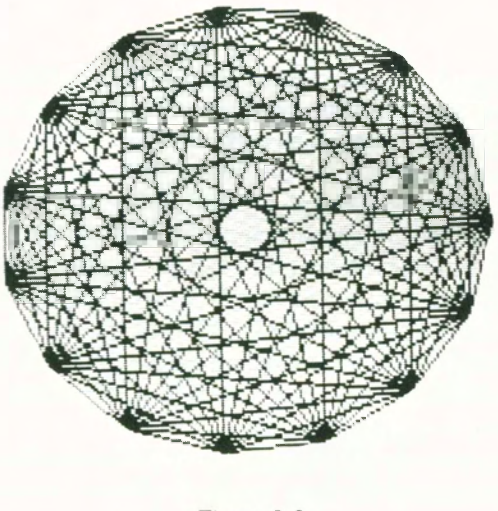


Figure 2.3

Example 2.3

Figure 2.3, a very popular design, is constructed by joining each vertex of a regular N-sided polygon (an N-gon) to every other vertex. N is not greater than 30.

We set the origin at the centre of the design, and all the vertices at a unit distance from the centre: the sizes of the HORIZ and VERT (3, 2.1), are chosen so that the design fits neatly on the screen. If one of these vertices lies on the positive x-axis (the horizontal), then the N vertices are all of the form (COS (ALPHA), SIN(ALPHA)), where ALPHA is an angle $2\pi I/N$ and I is chosen from 1, 2, . . ., or N. Here for the first time we see point coordinates being calculated by the program, not explicitly typed in, as in listing 2.6. Furthermore, since the program uses these values over and over again, it is sensible to store them in arrays and access them when required by specifying the correct array index. Note that in listing 2.9, if $1 \leqslant I \leqslant J \leqslant N$, then the J^{th} point is not joined to the I^{th} point; the line will have already been drawn in the opposite direction.

Listing 2.9

```
100 REM joining vertices of regular N-gon
110 LET start = 9700: LET setorigin = 9600: LET moveto = 9500
    : LET lineto = 9400
120 LET HORIZ = 3: LET VERT = 2.1
130 GO SUB start
140 LET XMOVE = HORIZ*0.5: LET YMOVE = VERT*0.5
150 GO SUB setorigin
160 DIM X(30): DIM Y(30)
169 REM setup vertices of regular N-gon in arrays X and Y.
170 INPUT "TYPE VALUE OF N ";N
180 LET ALPHA = 0: LET ADIF = 2*PI/N
190 FOR I = 1 TO N
200 LET X(I) = COS ALPHA: LET Y(I) = SIN ALPHA
210 LET ALPHA = ALPHA + ADIF
220 NEXT I
229 REM join point I to point J : 1<=I<J<=N.
230 FOR I = 1 TO N-1
240 FOR J = I+1 TO N
250 LET XPT = X(I): LET YPT = Y(I): GO SUB moveto
260 LET XPT = X(J): LET YPT = Y(J): GO SUB lineto
270 NEXT J
280 NEXT I
290 STOP
```

There are two immediate observations to be made from this very simple example. The first concerns *resolution*. Because the graphics frame is a discrete matrix, then *straight lines* must be approximated by a sequence of pixels. Unfortunately, the resolution of the Spectrum, like most microcomputer graphics systems, is low (that is, NXPIX and NYPIX are the order of hundreds) so the lines appear jagged; even in higher-resolution devices (like microfilm plotters) the same is true, but the sizes involved are so small that the jaggedness goes unnoticed.

The second observation is that as N increases in listing 2.9, the outline of the figure (the N-gon) approximates closely to a circle. Therefore we can use this idea to write a routine 'circle1' (listing 2.10a), which draws a circle with radius R about the centre (XCENT, YCENT) to give a picture similar to figure 2.4. Note that we are using angles measured in radians; that is, we are incrementing by 3/(R*XYSCALE) each time through the loop – a value that depends on the radius and produces a reasonable circle without waste of effort. Note also that since the vertices of the N-gon are only needed once, we do not store their values but calculate them as required. Again, the limitation in resolution of the screen is apparent on the circumference of the circle.

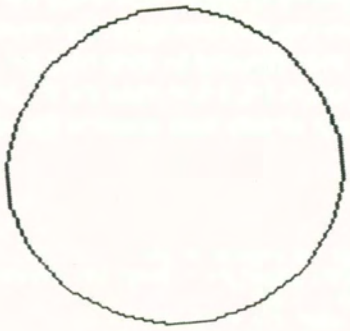


Figure 2.4

Listing 2.10a

```
300 REM circle1
301 REM IN  : XCENT, YCENT, R, XYSCALE
310 LET XMOVE = XCENT: LET YMOVE = YCENT : GO SUB setorigin
320 LET ADIF = 3/(R*XYSCALE)
330 LET XPT = R: LET YPT = 0: GO SUB moveto
339 REM calculate and join points (XPT,YPT) around the circle.
340 FOR A = ADIF TO 2*PI STEP ADIF
350 LET XPT = R*COS A: LET YPT = R*SIN A: GO SUB lineto
360 NEXT A
370 RETURN
```

Listing 2.10b

```
400 REM circle2
401 REM IN  : XCENT, YCENT, R, XYSCALE
410 CIRCLE FN X(XCENT),FN Y(YCENT),R*XYSCALE
420 RETURN
```

We saw that the Spectrum has a BASIC function CIRCLE that enables us to draw a circle. So we can incorporate this in a primitive routine 'circle2' (listing 2.10b) for drawing a circle, one that is necessarily more efficient than 'circle1'.

Whenever we use such routines, we must be aware of any *side-effects* produced; for example, has the origin or plot head been moved by the routine? For example, listing 2.10a changes the position of both the origin and plot pen, whereas listing 2.10b does not. It would therefore be sensible to add the following line to the 'circle1' routine

```
370 LET XMOVE=–XCENT: LET YMOVE=–YCENT: GO SUB setorigin : RETURN
```

Exercise 2.4
Write a routine to draw an ellipse of major axis A units (horizontal) and minor axis B units (vertical). Note that a typical point on this ellipse has coordinates (A cos α, B sin α) where $0 \leqslant \alpha \leqslant 2\pi$. However, it must be remembered that, unlike the circle, α is not the angle made by the radius through the point with the positive x-axis. It is simply a descriptive parameter.

Incorporate this routine in a program that draws a diagram similar to figure 2.5. Here are two things to note: (1) there is no need for A to be greater than B; and (2) observe the optical illusion of the two apparent white diagonal lines. Another illusion can be seen in figure 2.3 – dark circles radiating out from the centre of the pattern. The study of optical illusions is fascinating (see Tolansky, 1964) and it is a never-ending source of ideas for patterns. This exercise was introduced because it leads the way to the general technique of drawing curves (see chapters 3 and 6).

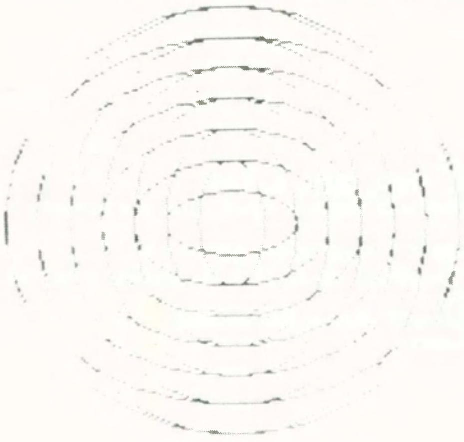


Figure 2.5

Example 2.4
An extension of this idea, the natural next step, is the construction of a spiral. Again the general form of the curve about the origin is (R cos α, R sin α) but now α varies between angles β to $\beta + 2N\pi$, where β (the parameter BETA) is the initial angle that the normal to the spiral makes with the positive x-axis, and N is the

number of turns in the spiral. The radius R is no longer a constant value, but varies with the value of α: if RMAX is the outer radius of the spiral then R is given by the formula

$$R = \text{RMAX}(\alpha - \beta)/2N\pi$$

Note that this routine, which centres the spiral at (XCENT, YCENT), causes no side-effects because we reset the origin back to its original position before leaving the routine.

Listing 2.11a

```
300 REM spiral1
301 REM IN   : XCENT, YCENT, RMAX, N, BETA
310 LET XMOVE = XCENT: LET YMOVE = YCENT : GO SUB setorigin
320 LET ADIF = PI/50: LET ALPHA = BETA
330 LET RDIF = RMAX/(N*100)
339 REM calculate and join points (XPT,YPT) on the spiral.
340 FOR R = RDIF TO RMAX STEP RDIF
350 LET XPT = R*COS ALPHA: LET YPT = R*SIN ALPHA: GO SUB lineto
360 LET ALPHA = ALPHA + ADIF
370 NEXT R
380 LET XMOVE = -XCENT: LET YMOVE = -YCENT : GO SUB setorigin
390 RETURN
```

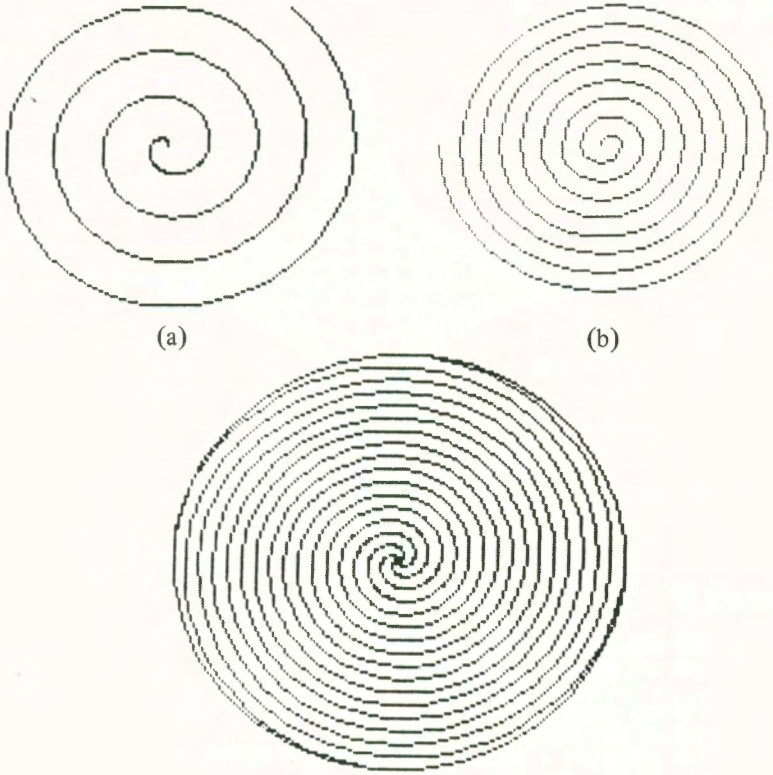
Listing 2.11b

```
300 REM celtic/spiral
301 REM IN   : XCENT, YCENT, RMAX, N, SIGN
310 LET XMOVE = XCENT: LET YMOVE = YCENT : GO SUB setorigin
320 PLOT XPEN,YPEN
330 LET R = 0: LET RDIF = RMAX/(N*2): LET S = 1
339 REM construct the spiral using DRAW to produce a series of semicircles.
340 FOR I = 1 TO N*2
350 LET R = R + RDIF: LET XPIX = S*R*XYSCALE
360 DRAW XPIX,0,SIGN*PI
370 LET S = -S
380 NEXT I
390 RETURN
```

Exercise 2.5

Listing 2.11a produces a diagram similar to figure 2.6a (with XCENT = 0, YCENT = 0, N = 4, BETA = 1 and RMAX = 3). What happens if you set RMAX to –3? Use the routine in a program that generates figure 2.6c. Again note the optical illusion when the observer's head is moved in a circle in front of the diagram, keeping the horizontal (and hence also the vertical) direction parallel with the original. The spirals appear to rotate about the centre!

Example 2.5
Spirals have been used in art and design for thousands of years; however, most of the ancient spirals were not true spirals but consisted of sequences of semicircles (see Bain, 1972). Listing 2.11b (a routine with input parameters RMAX, N and SIGN – the orientation value being ±1) enables us to draw such semicircular spirals using the DRAW option, and so it is much more efficient than the accurate method of listing 2.11a (for example, figure 2.6b).


(a) (b)

(c)

Figure 2.6

Exercise 2.6
Follow the logic of listing 2.11b, and extend it so that the normal to the original curve does not go along the x-axis but makes an angle BETA with it. In the Book of Kells there are examples of *triskeles*, which are composed of a set of third-circles joined in sequence. Experiment in constructing these and any other variations on this method, such as quarter-circles, etc.

Example 2.6
Write a routine (listing 2.12) that draws diagrams similar to figure 2.7.

Here we introduce the concept of an *envelope*. Instead of drawing a curve by a sequence of small line segments (as in the circle of listing 2.9), we devise a sequence of lines that are tangential to the curve. For example, the figure shows four rectangular hyperbolae placed in the *quarters* of the plane.

N points are placed on each of the four arms (of unit length) that divide the plane into the four quarters. The 4N points are therefore (±I/N, 0.0) and (0.0, ±I/N) where I = 1, 2, . . , N.

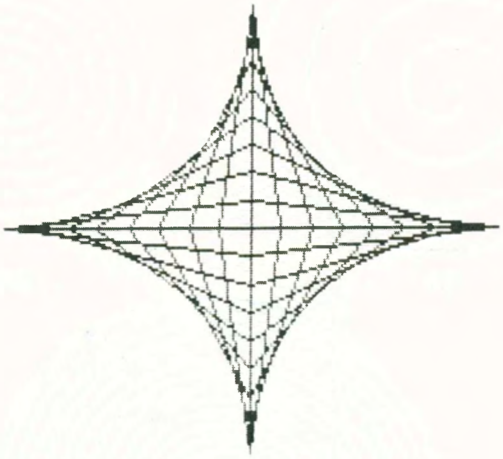


Figure 2.7

Listing 2.12

```
100 REM example of an envelope
110 LET start = 9700: LET setorigin = 9600
    : LET moveto = 9500: LET lineto = 9400
120 LET HORIZ = 3: LET VERT = 2.1
130 GO SUB start
140 LET XMOVE = HORIZ*0.5: LET YMOVE = VERT*0.5
150 GO SUB setorigin
159 REM draw unit axes in graphics area.
160 INPUT "TYPE N ";N
170 LET XPT =  1: LET YPT =  0: GO SUB moveto
180 LET XPT = -1: LET YPT =  0: GO SUB lineto
190 LET XPT =  0: LET YPT =  1: GO SUB lineto
200 LET XPT =  0: LET YPT = -1: GO SUB lineto
208 REM produce N sets each of four points, one on each axis.
209 REM join the points of each set in order.
210 FOR I = 1 TO N
220 LET ID1 = I/N: LET ID2 = (N + 1 - I)/N
230 LET XPT =  ID1: LET YPT =    0: GO SUB moveto
240 LET XPT =    0: LET YPT =  ID2: GO SUB lineto
250 LET XPT = -ID1: LET YPT =    0: GO SUB lineto
260 LET XPT =    0: LET YPT = -ID2: GO SUB lineto
270 LET XPT =  ID1: LET YPT =    0: GO SUB lineto
280 NEXT I
290 STOP
```

Exercise 2.7
Generalise this routine so that there is a variable number of arms, M, stretching out from the origin and dividing the plane into equal segments.

Exercise 2.8
Draw a diagram similar to figure 2.8; the routine will have an integer parameter N. It will calculate 4N points $\{P(I): I = 1, 2, \ldots, 4N\}$ around the edges of a square of unit side, starting at a corner. There is one point at each corner and the points are placed so that the distance between consecutive points is 1/N. Then, pairs of points are joined according to the following rule: P(I) is joined to P(J) for all positive I and J less than or equal to 4N, such that J – I (subtraction modulo 4N) belongs to the sequence 1, 1 + 2, 1 + 2 + 3, . . . For example, if N is 10, then P(20) is joined to P(21), P(23), P(26), P(30), P(35), P(1), P(8) and P(16). The outer square should be drawn, and hence if two points lie on the same side of the square there is no need to join them by a line since it already exists as an edge of the outer square. For example, P(20) is a corner, so it is on the same edge as P(16) and also P(21), P(23), P(26) and P(30).

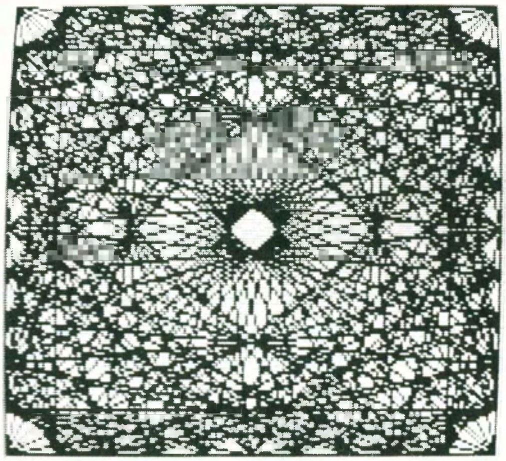


Figure 2.8

Example 2.7
Emulate a Spirograph, in order to produce diagrams similar to figure 2.9.

A Spirograph consists of a cogged disc inside a cogged circle, which is placed on a piece of paper. Let the outer circle have integer radius A and the disc integer radius B. The disc is always in contact with the circle. There is a small hole in the disc at a distance D (also an integer) from the centre of the disc, through which is placed a sharp pencil point. The disc is moved around the circle in an anti-clockwise direction, but it must always touch the outer circle; the cogs ensure

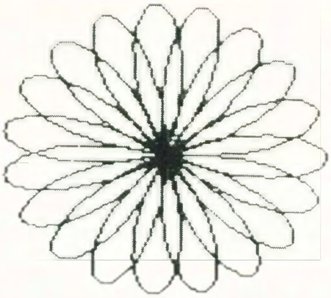


Figure 2.9

that there is no slipping. The pencil point traces out a pattern, which is complete when the pencil returns to its original position.

Initially we assume that the centres of the disc and the circle and also the hole all lie on the positive x-axis, the centre of the circle being the coordinate origin. In order to emulate the Spirograph we need to specify a general point on the track of the pencil point. We let α be the angle made with the positive x-axis by the line joining the origin to the point where the circle and disc touch. The point of contact is therefore $(A \cos \alpha, A \sin \alpha)$ and the centre of the disc is $((A - B) \cos \alpha, (A - B) \sin \alpha)$. If we let β be the angle that the line joining the hole to the centre of the disc makes with the x-direction, then the coordinates of the hole are

$$((A - B) \cos \alpha + D \cos \beta, (A - B) \sin \alpha + D \sin \beta)$$

The point of contact between the disc and circle will have moved through a distance $A\alpha$ around the circle, and a distance $-B\beta$ around the disc (the minus sign is because α and β have the opposite orientation). Since there is no slipping, these distances must be equal, and hence we have the equation $\beta = -(A/B)\alpha$. The pencil returns to its original position when both α and β are integer multiples of 2π. When $\alpha = 2N\pi$, then $\beta = -N(A/B)2\pi$: hence the pencil point returns to its original position for the first time when $N(A/B)$ becomes an integer for the first time; that is, when N is equal to B divided by the highest common factor of B and A. The routine 'Euclid' (listing 2.13) uses Euclid's algorithm (see Davenport, 1952) to calculate the highest common factor (integer HCF) of two positive integers A and B.

This function is used in the routine 'spiro' (listing 2.13), which calculates the value of N and then varies α (ALPHA) between 0 and $2N\pi$ in steps of $\pi/100$; for each α, the value of β (BETA) is calculated and then the general track is drawn. Figure 2.9 was drawn by a call to 'spiro' with A = 12, B = 7 and D = 5. The size of HORIZ and VERT must be chosen so that the figure fits on the screen; in this case HORIZ = 30 and VERT = 20.

Listing 2.13

```
200 REM Euclid
201 REM IN  : A, B
202 REM OUT : HCF
210 LET I = A: LET HCF = B
220 IF A < B THEN LET I = B: LET HCF = A
230 LET J = I - INT (I/HCF)*HCF
240 IF J = 0 THEN RETURN
250 LET I = HCF: LET HCF = J: GO TO 230

300 REM spiro
301 REM IN  : A, B, D
310 LET RAB = A - B: LET ALPHA = 0: LET ADIF = PI/50: LET AOB = A/B
320 GO SUB Euclid: LET N = B/HCF: LET NC = 100*N
330 LET XPT = RAB + D: LET YPT = 0: GO SUB moveto
339 REM calculate and join points (XPT,YPT) on the path of a Spirograph.
340 FOR I = 1 TO NO
350 LET ALPHA = ALPHA + ADIF
360 LET BETA = ALPHA*ACB
370 LET XPT = RAB*COS ALPHA + D*COS BETA
380 LET YPT = RAB*SIN ALPHA - D*SIN BETA
390 GO SUB lineto
400 NEXT I
410 RETURN
```

It is evident from this example that drawing patterns is not so straightforward as it appears. Even such a simple picture as figure 2.8 requires the mathematical backup of Euclid. Progressing through computer graphics, we shall discover more and more that it is essential to have at least an elementary knowledge of not only coordinate geometry but also calculus, algebra, Euclidean geometry and number theory. Be prepared to scour your local library (or pester your friendly neighbourhood mathematician) for the necessary information.

Complete Programs

At this stage we shall group the listings 2.1 ('start'), 2.2 (two functions FN X and FN Y), 2.3 ('setorigin'), 2.4 ('moveto') and 2.5 ('lineto') under the heading 'lib1'. Later we shall replace listing 2.5 with listing 3.3 ('clip' and a new version of 'lineto').

I. 'lib1' and listing 2.6 ('drawing a square'): no INPUT data.
II. 'lib1' and listing 2.7 ('main program' and 'polygon'): requires the number of vertices on a polygon, and their X/Y coordinates in pairs ($-15 \leqslant X \leqslant 15$ and $-10 \leqslant Y \leqslant 10$).
III. 'lib1' and listing 2.9 ('joining vertices of regular N-gon'): requires an integer $N \leqslant 30$.
IV. 'lib1' and your own 'main program' (listing 2.7 will be useful as a model) calling listings 2.10a ('circle1') and 2.10b ('circle2'). Each routine requires

the centre (XCENT, YCENT) and radius R. Choose these values so that the figure is consistent with your values of HORIZ, VERT, XMOVE and YMOVE. Try 30, 20, 15 and 10 respectively. As an example call 'circle1' with centre (1, –1), radius 8 and 'circle2' with centre (1, 2), radius 5.

V. 'lib1' and your own 'main program' calling listings 2.11a ('spiral1') and 2.11b ('celtic'). Each routine requires the centre (XCENT, YCENT), maximum radius RMAX and number of turns in the spiral N. Listing 2.11a ('spiral1') also requires an angle BETA, whereas 2.11b ('celtic') needs a value SIGN, which is ±1. Choose these values so that the figure is consistent with your values of HORIZ, VERT, XMOVE and YMOVE (for example, 30, 20, 15 and 10). For example, call 'spiral1' with centre (1, –1), RMAX = 8, N = 3 and BETA = 2, and call 'celtic' with centre (1, 2), RMAX = 5, N = 5 and SIGN = –1. Also try SIGN = +1.

VI. 'lib1' and listing 2.12 ('envelope'): requires a integer N, $2 \leqslant N \leqslant 30$.

VII. 'lib1' and listing 2.13 ('Euclid' and 'spiro'): requires three integers A, B and D, where $A > B > D$. Choose HORIZ, VERT, etc., so that the diagram fits on the screen: set XMOVE = HORIZ*0.5, YMOVE = 0.5*VERT (for 'setorigin'), where both HORIZ and VERT are greater than 2*(A – B + D).

3 *Two-dimensional Coordinate Geometry*

In chapter 2 we introduced the concept of the two-dimensional rectangular co-ordinate system; we defined points in space as vectors, from which we were able to draw line segments between pairs of points. To be strictly accurate, a *straight line* (or line for short) in two-dimensional space is not a finite segment, but stretches off to infinity in both directions, and so we need to introduce ways of representing a general point on such a line.

We are taught that the equation of a straight line is $y = mx + c$, the relationship between the x-coordinate and y-coordinate of a general point on the line, where m is the tangent of the angle that the line makes with the positive x-axis, and c is the point of intersection of the line with the y-axis; that is, when $x = 0$ then $y = c$. This formula may be well known, but it is not very useful. What happens if the line is vertical? m is infinite! A far better formula is

$$ay = bx + c$$

This allows for all possible lines: if the line is vertical, a is 0; (b/a) is now the tangent of the angle that the line makes with the positive x-axis, and the line cuts the y-axis at (c/a), provided that a is not equal to zero, and the x-axis at $(-c/b)$, provided that b is not equal to zero. The line is parallel to the y-axis if a is zero, and to the x-axis if b is zero.

We shall frequently use this formulation of a line in the following pages; however, we now introduce another, possibly more useful, method for defining a line. Before we can describe this new method we must first define two operations on vectors (namely, scalar multiple and vector addition), as well as describe another required operation – the absolute value of a vector. Suppose we have two vectors $\boldsymbol{p}_1 \equiv (x_1, y_1)$ and $\boldsymbol{p}_2 \equiv (x_2, y_2)$, then
scalar multiple $k\boldsymbol{p}_1 = (k \times x_1, k \times y_1)$, we multiply the individual coordinates by some scalar value (that is, real) k.

vector addition $\boldsymbol{p}_1 + \boldsymbol{p}_2 = (x_1 + x_2, y_1 + y_2)$, add the x-coordinates together, and the y-coordinates together.
absolute value $|\boldsymbol{p}_1| = \sqrt{(x_1^2 + y_1^2)}$ is the distance of the point $\boldsymbol{p}_1$ from the origin (this is also called the length, and the amplitude of the vector).

To define a line we first arbitrarily choose any two points on the line, again we call them $\boldsymbol{p}_1 \equiv (x_1, y_1)$ and $\boldsymbol{p}_2 \equiv (x_2, y_2)$. A general point $\boldsymbol{p}(\mu) \equiv (x, y)$ is given by the combination of scalar multiples and vector addition

$$(1 - \mu)\boldsymbol{p}_1 + \mu \boldsymbol{p}_2 \qquad \text{for some real value of } \mu$$

that is, the vector $((1 - \mu) \times x_1 + \mu \times x_2, (1 - \mu) \times y_1 + \mu \times y_2)$. We place the μ in brackets after $\boldsymbol{p}$ to show the dependence of the vector on the value of μ. Later when we understand the relationship more fully we shall leave out the (μ). If $0 \leqslant \mu \leqslant 1$, then $\boldsymbol{p}(\mu)$ lies on the line somewhere between $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$. For any specified point $\boldsymbol{p}(\mu)$, the value of μ is given by the ratio

$$\frac{\text{distance of } \boldsymbol{p}(\mu) \text{ from } \boldsymbol{p}_1}{\text{distance of } \boldsymbol{p}_2 \text{ from } \boldsymbol{p}_1}$$

where the measure of distance is positive if $\boldsymbol{p}(\mu)$ is on the same side of $\boldsymbol{p}_1$ as $\boldsymbol{p}_2$, and negative otherwise. The positive distance between any two vector points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ is given by (Pythagoras)

$$|\boldsymbol{p}_2 - \boldsymbol{p}_1| = \sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)}$$

See figure 2.1, which shows a line segment between points $(-3, -1) \equiv \boldsymbol{p}(0)$ and $(3, 2) \equiv \boldsymbol{p}(1)$: the point $(1, 1)$ lies on the line as $\boldsymbol{p}(2/3)$. Note that $(3, 2)$ is a distance $3\sqrt{5}$ from $(-3, -1)$, whereas $(1, 1)$ is a distance $2\sqrt{5}$. From now on we omit the (μ) from the point vector.

Example 3.1
We can further illustrate this idea by drawing the pattern shown in figure 3.1. At first sight it looks complicated, but on closer inspection it is seen to be simply a square, outside a square, outside a square, etc. The squares are getting successively smaller and they are rotating through a constant angle. In order to draw the diagram we need a technique that, when given a general square, draws a smaller internal square rotated through this fixed angle. Suppose the general square has four corners $\{(x_i, y_i) | i = 1, 2, 3, 4\}$ and the ith side of the square is the line joining (x_i, y_i) to (x_{i+1}, y_{i+1}), assuming additions of subscripts are modulo 4 (that is, 4 + 1 = 1). A general point on this side of the square, (x'_i, y'_i), is given by

$$((1 - \mu) \times x_i + \mu \times x_{i+1}, (1 - \mu) \times y_i + \mu \times y_{i+1}) \quad \text{where} \quad 0 \leqslant \mu \leqslant 1$$

In fact $\mu:1 - \mu$ is the ratio in which the side is bisected. If μ is fixed and the four points $\{(x'_i, y'_i) | i = 1, 2, 3, 4\}$ are calculated in the above manner, then the sides of the new square make an angle $\alpha = \tan^{-1} [\mu/(1 - \mu)]$ with the corresponding side of the outer square. So, by keeping μ fixed for each new square, the angle between consecutive squares remains a constant α. In listing 3.1, which generated figure 3.1, there are 21 squares and $\mu = 0.1$.

Listing 3.1

```
100 REM square outside square etc.
110 LET start = 9700: LET setorigin = 9600: LET moveto = 9500
    : LET lineto = 9400
120 LET HORIZ = 3: LET VERT = 2.1
130 GO SUB start
140 LET XMOVE = HORIZ*0.5: LET YMOVE = VERT*0.5
150 GO SUB setorigin
160 DIM X(4): DIM Y(4): DIM V(4): DIM W(4)
170 DATA 1,1,1,-1,-1,-1,-1,1
179 REM initialise first square.
180 FOR I = 1 TO 4: READ X(I),Y(I): NEXT I
189 REM set MU value and draw 20 squares.
190 LET MU = 0.1: LET UM = 1 - MU
200 FOR I = 1 TO 21
208 REM join four vertices of square (X(J),Y(J)) : J=1:4.
209 REM calculate next four vertices (V(J),W(J)) : J=1:4.
210 LET XPT = X(4): LET YPT = Y(4): GO SUB moveto
220 FOR J = 1 TO 4
230 LET XPT = X(J): LET YPT = Y(J): GO SUB lineto
240 LET NJ = J + 1: IF NJ = 5 THEN LET NJ = 1
250 LET V(J) = UM*X(J) + MU*X(NJ)
260 LET W(J) = UM*Y(J) + MU*Y(NJ)
270 NEXT J
279 REM copy arrays V and W into X and Y.
280 FOR J = 1 TO 4
290 LET X(J) = V(J): LET Y(J) = W(J)
300 NEXT J
310 NEXT I
320 STOP
```

It is useful to note that the vector combination form of a line can be reorganised

$$\boldsymbol{p}_1 + \mu(\boldsymbol{p}_2 - \boldsymbol{p}_1)$$

When given in this new representation the vector $\boldsymbol{p}_1$ can be called the *base vector*, and $(\boldsymbol{p}_2 - \boldsymbol{p}_1)$ can be called the *directional vector*. In fact any point on the line can stand as a base vector; it simply acts as a point to anchor a line that is parallel to the directional vector. This concept of a vector acting as a direction needs some further explanation. We have already seen that a vector pair, (x, y) say, may represent a point; a line joining the coordinate origin to this point may be thought of as specifying a direction – any line in space that is parallel to this line is defined to have the same directional vector. We insist that the line goes

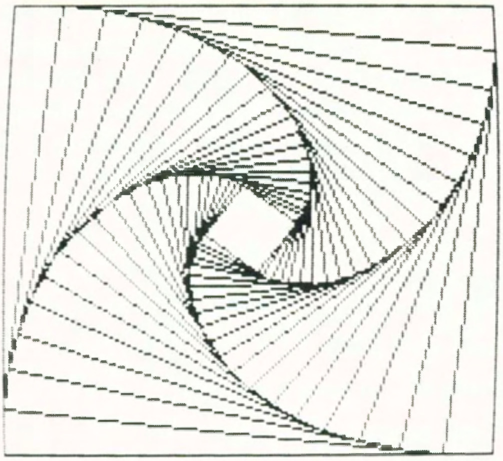


Figure 3.1

from the origin towards (x, y), the so-called positive *sense*; a line from (x, y) towards the origin has negative sense.

This dual interpretation of a vector, as a point or a direction, is used in the following example.

Example 3.2
Draw a dashed line, with 13 dashes (and hence 12 spaces between dashes) from point $\boldsymbol{p}_1 \equiv (x_1, y_1)$ to $\boldsymbol{p}_2 \equiv (x_2, y_2)$. This problem is solved by finding the 26 equi-spaced points on the line; that is $\boldsymbol{p}_1 + i/25\,(\boldsymbol{p}_2 - \boldsymbol{p}_1)$ where i varies from 0 (at $\boldsymbol{p}_1$) to 25 (at $\boldsymbol{p}_2$). We draw consecutively or move between neighbouring points using the CalComp 'plot' (listing 2.8). There is no need to store the values; we already have $\boldsymbol{p}_1$, so, by adding $1/25(\boldsymbol{p}_2 - \boldsymbol{p}_1)$ each time, we can move through all the required points (see listing 3.2).

Listing 3.2

```
100 REM dashed lines
110 LET start = 9700: LET plot = 9800
120 LET HORIZ = 3: LET VERT = 2.1
130 GO SUB start
140 LET XPT = HORIZ*0.5: LET YPT = VERT*0.5: LET MODE = -3
150 GO SUB plot
160 INPUT "TYPE X1 AND Y1  ";X1;" , ";Y1
170 INPUT "TYPE X2 AND Y2  ";X2;" , ";Y2
179 REM move to first point.
180 LET XPT = X1: LET YPT = Y1: LET MODE = 3: GO SUB plot
190 LET XD = (X2 - X1)/25: LET YD = (Y2 - Y1)/25
199 REM alternately draw and move to next 25 points on the line.
200 FOR I = 1 TO 25
210 LET XPT = XPT + XD: LET YPT = YPT + YD: LET MODE = 5 - MODE: GO SUB plot
220 NEXT I
230 STOP
```

Exercise 3.1
Experiment by drawing different types of dashed lines: for example, (a) the size of the dash could be twice that of the space between; (b) the size of the dash could be a fixed numerical value and the number of dashes unknown; (c) the dashes could vary in size, alternating between large and short dashes, where the relationship between the types of dashes and the spaces could be input variables.

This base and direction representation is also very useful for calculating the point of intersection of two lines, a problem that frequently crops up in two-dimensional graphics. Suppose we have two lines $\boldsymbol{p} + \mu\boldsymbol{q}$ and $\boldsymbol{r} + \lambda\boldsymbol{s}$, where $\boldsymbol{p} \equiv (x_1, y_1), \boldsymbol{q} \equiv (x_2, y_2), \boldsymbol{r} \equiv (x_3, y_3)$ and $\boldsymbol{s} \equiv (x_4, y_4)$ for $-\infty < \mu, \lambda < \infty$. We need to find the unique values of μ and λ such that

$$\boldsymbol{p} + \mu\boldsymbol{q} = \boldsymbol{r} + \lambda\boldsymbol{s}$$

that is, a point that is common to both lines. This vector equation can be written as two separate equations

$$x_1 + \mu \times x_2 = x_3 + \lambda \times x_4 \tag{3.1}$$

$$y_1 + \mu \times y_2 = y_3 + \lambda \times y_4 \tag{3.2}$$

Rewriting these equations we get

$$\mu \times x_2 - \lambda \times x_4 = x_3 - x_1 \tag{3.3}$$

$$\mu \times y_2 - \lambda \times y_4 = y_3 - y_1 \tag{3.4}$$

Multiplying (3.3) by y_4, (3.4) by x_4 and subtracting we get

$$\mu \times (x_2 \times y_4 - y_2 \times x_4) = (x_3 - x_1) \times y_4 - (y_3 - y_1) \times x_4$$

If $(x_2 \times y_4 - y_2 \times x_4) = 0$ then the lines are parallel and there is no point of intersection (μ does not exist), otherwise

$$\mu = \frac{(x_3 - x_1) \times y_4 - (y_3 - y_1) \times x_4}{(x_2 \times y_4 - y_2 \times x_4)} \tag{3.5}$$

and similarly

$$\lambda = \frac{(x_3 - x_1) \times y_2 - (y_3 - y_1) \times x_2}{(x_2 \times y_4 - y_2 \times x_4)} \tag{3.6}$$

The solution becomes even simpler if one of the lines is parallel to a coordinate axis. Suppose this line is $x = d$, then we can set $\boldsymbol{r} \equiv (d, 0)$ and $\boldsymbol{s} \equiv (0, 1)$, which when substituted in equation (3.5) gives

$$\mu = (d - x_1)/x_2$$

and similarly if the line is $y = d$

$$\mu = (d - y_1)/y_2$$

Naturally if both lines are parallel then the denominator in these equations becomes zero and we get an infinite result, because the two parallel lines do not intersect.

Example 3.3
Find the point of intersection of the two lines (a) joining (1, –1) to (–1, –3) and (b) joining (1, 2) to (2, –2).

The lines may be written

$$(1 - \mu)(1, -1) + \mu(-1, -3) \quad -\infty < \mu < \infty \tag{3.7}$$

$$(1 - \lambda)(1, 2) + \lambda(2, -2) \quad -\infty < \lambda < \infty \tag{3.8}$$

or when placed in the base/directional vector form

$$(1, -1) + \mu(-2, -2) \tag{3.9}$$

$$(1, 2) + \lambda(2, -4) \tag{3.10}$$

Substituting these values in equation (3.5) gives

$$\mu = \frac{(1 - 1) \times -4 - (2 + 1) \times 2}{(-2 \times -4 - (-2) \times 2)} = -1/2$$

whence the point of intersection is $(1, -1) - 1/2(-2, -2) \equiv (2, 0)$.

Exercise 3.2
Experiment with this concept of vector representation of two-dimensional space. You can make up your own questions: it is easy to check that your answers are correct. Consider example 3.2. We know that (2, 0) lies on the first line because we used the value $\mu = -1/2$: our answer is correct if it álso lies on the second line; it does with $\lambda = 1/2$.

Exercise 3.3
Write a program that reads in data about two straight lines (it can be either in the form of equations, or in the base/directional vector form) and then calculates their point of intersection (if any).

Clipping

By now you will have realised that it is impossible to PLOT, or DRAW, to a pixel (x, y) outside the graphics area, and thus we are limited to $0 \leqslant x \leqslant 255$ and $0 \leqslant y \leqslant 175$. It is far too easy to stray inadvertently outside this area. In fact when drawing two-dimensional and three-dimensional scenes it is commonplace to define scenes that cover an area greater than that allocated to graphics on the Spectrum. So it is necessary to find an algorithm that will *clip* off all exterior line segments without losing any that should be drawn.

We assume that the centre of the screen is given by the (non-pixel) point $(255/2, 175/2) \equiv (127.5, 87.5)$ and thus the four corners of the graphics area are $(127.5 \pm 127.5, 87.5 \pm 87.5)$. Our problem reduces to calculating which part (if any) of a line segment joining pixel point (XA, YA) to pixel point (XB, YB) lies within the area. In order to simplify matters we redefine our pixel coordinate system to let the centre of the screen be the origin, by subtracting the vector (127.5, 87.5) from the coordinates of original points. The graphics rectangle now has corners $(\pm 127.5, \pm 87.5)$. We extend the sides of the rectangle, thus dividing space into nine sectors; see figure 3.2, which also shows the graphics area and the BORDER. In this diagram a number of different line segments have been drawn to aid the explanation of the algorithm. Each point in space may now be classified by two parameters IX and IY where

(1) IX = −1, 0 or +1 depending on whether the x-coordinate value of the point lies to the left, on or to the right of the graphics area;
(2) IY = −1, 0 or +1 depending on whether the y-coordinate of the point lies below, on or above the graphics rectangle.

These values are calculated, when needed, inside the algorithm program.

If the two points at the end of the line segment – that is, (XA, YA) and (XB, YB) – have parameters IXA and IYA, and IXB and IYB respectively, then there are a number of possibilities to consider.
(i) If IXA = IXB ≠ 0 or IYA = IYB ≠ 0, then the whole line segment is outside the rectangle and hence may be safely ignored; for example, line AB in figure 3.2.
(ii) If IXA = IYA = IXB = IYB = 0, then the whole line segment lies in the graphics area and so the complete line must be drawn; for example, line CD.
(iii) The remaining case must be considered in detail. If IXA ≠ 0 and/or IYA ≠ 0 then the point (XA, YA) lies outside the rectangle and so new values for XA and YA must be found – to avoid confusion we will call these XA′ and YA′. (XA′, YA′) is the point on the line segment nearer to (XA, YA) where the line cuts the graphics area. The formula for this calculation was considered above; that is, the intersection of a line with another line parallel to a coordinate axis. If the line misses the rectangle, then we define (XA′, YA′) to be that point where the line

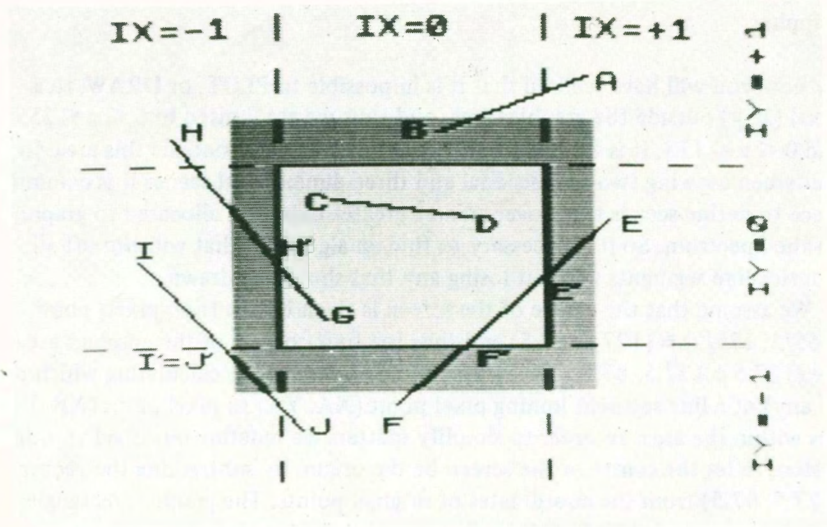


Figure 3.2

cuts one of the extended vertical edges. If IXA = IYA = 0 then (XA′, YA′) ≡ (XA, YA). The point (XB′, YB′) is calculated in a similar manner; see the algorithm given by routine 'clip' in listing 3.3. The required clipped line is that joining (XA′, YA′) to (XB′, YB′). If the original line misses the rectangle then the algorithm ensures that (XA′, YA′) = (XB′, YB′) and the new line segment degenerates to a point and is ignored. For example, EF is clipped to E′F′, GH is clipped to GH′ (G = G′) and IJ degenerates to a point I′ = J′.

Thus 'clip' takes the two pixel end points of the line, (XA, YA) and (XB, YB), and transforms them into the centred system. It then discovers which of the above three possibilities is relevant and deals with it thus: (i) exit the routine immediately; (ii) join the two points; or (iii) calculate the 'dashed' points and join them with a line.

Listing 3.3 also includes a new version of 'lineto' routine that calls 'clip' instead of PLOT and DRAW, thus enabling it to cope with the problem of joining lines anywhere in space. From now on always use this new version of 'lineto'. It will prove invaluable, especially in the study of three-dimensional objects.

Exercise 3.4
Use this altered routine in the programs of chapter 2. Choose values of HORIZ and VERT in such a way that some lines in the diagrams go outside the graphics area.

Listing 3.3

```
8400 REM clip
8401 REM IN  : XA,YA,XB,YB
8409 REM change coordinate system.
8410 LET XA = XA - 127.5: LET YA = YA - 87.5: LET XB = XB - 127.5
     : LET YB = YB  - 87.5
8419 REM find the sector values of two points (XA,YA) AND (XB,YB).
8420 LET IXA = 0: IF ABS XA > 127.5 THEN LET IXA = SGN XA
8430 LET IYA = 0: IF ABS YA > 87.5 THEN LET IYA = SGN YA
8440 LET IXB = 0: IF ABS XB > 127.5 THEN LET IXB = SGN XB
8450 LET IYB = 0: IF ABS YB > 87.5 THEN LET IYB = SGN YB
8459 REM points in same off-screen sector then return.
8460 IF IXA*IXB = 1 OR IYA*IYB = 1 THEN RETURN
8470 IF IXA = 0 THEN GO TO 8500
8479 REM move 1'st point to nearer x-edge.
8480 LET XX = 127.5*IXA: LET YA = YA + (YB - YA)*(XX - XA)/(XB - XA)
     : LET XA = XX
8490 LET IYA = 0: IF ABS YA > 87.5 THEN LET IYA = SGN YA
8500 IF IYA = 0 THEN GO TO 8515
8509 REM move 1'st point to nearer y-edge.
8510 LET YY = 87.5*IYA: LET XA = XA + (XB - XA)*(YY - YA)/(YB - YA)
     : LET YA = YY
8515 IF ABS (XA - XB) < 0.000001 AND ABS (YA - YB) < 0.000001 THEN RETURN
8520 IF IXB = 0 THEN GO TO 8550
8529 REM move 2'nd point to nearer x-edge.
8530 LET XX = 127.5*IXB: LET YB = YA + (YB - YA)*(XX - XA)/(XB - XA)
     : LET XB = XX
8540 LET IYB = 0: IF ABS YB > 87.5 THEN LET IYB = SGN YB
8550 IF IYB = 0 THEN GO TO 8570
8559 REM move 2'nd point to nearer y-edge.
8560 LET YY = 87.5*IYB: LET XB = XA + (XB - XA)*(YY - YA)/(YB - YA)
     : LET YB = YY
8570 IF ABS (XA - XB) < 0.000001 AND ABS (YA - YB) < 0.000001 THEN RETURN
8579 REM plot non-coincident points.
8580 LET XA = INT (XA + 128): LET YA = INT (YA + 88)
     : LET XB = INT (XB + 128): LET  YB = INT (YB + 88)
8590 PLOT XA,YA: DRAW XB - XA,YB - YA: RETURN

9400 REM lineto/ clipping
9401 REM IN  : XPT,YPT,XPEN,YPEN
9401 REM OUT : XPEN,YPEN
9410 LET XA=XPEN: LET YA=YPEN
9420 LET XPEN=FN X(XPT)
9430 LET YPEN=FN Y(YPT)
9440 LET XB=XPEN: LET YB=YPEN
9450 GO SUB clip
9460 RETURN
```

Returning to the use of a vector ($\boldsymbol{q} \equiv (x, y) \neq (0, 0)$, say) representing a direction, we note that any positive scalar multiple $k\boldsymbol{q}$, for $k > 0$, represents the *same direction and sense* as $\boldsymbol{q}$. (If k is negative then the direction has its sense inverted). In particular, setting $k = 1/|\boldsymbol{q}|$ produces a vector $(x/\sqrt{(x^2 + y^2)}, y/\sqrt{(x^2 + y^2)})$ with unit absolute value.

Thus a general point on a line, $\boldsymbol{p} + \mu\boldsymbol{q}$, is a distance $|\mu\boldsymbol{q}|$ from the base point $\boldsymbol{p}$, and if $|\boldsymbol{q}| = 1$ (a unit vector) then the point is a distance $|\mu|$ from $\boldsymbol{p}$.

We now consider the angles made by directional vectors with various fixed directions. Suppose that α is the angle between the line joining $\boldsymbol{O}$ (the origin) to $\boldsymbol{q} \equiv (x, y)$, and the positive x-axis. Then $x = |\boldsymbol{q}| \times \cos\alpha$ and $y = |\boldsymbol{q}| \times \sin\alpha$; see figure 3.3 – there are similar figures for the three other quadrants.

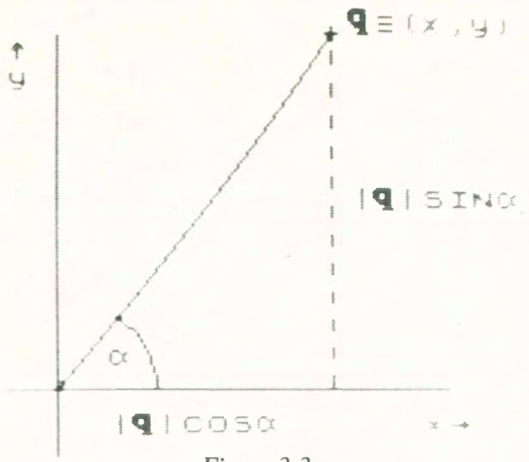


Figure 3.3

If q is a unit vector (that is, $|q| = 1$) then $q \equiv (\cos \alpha, \sin \alpha)$. We note that $\sin \alpha = \cos (\alpha - \pi/2)$ for all values of α. Thus we can rewrite $q = (\cos \alpha, \cos (\alpha - \pi/2))$, but $\alpha - \pi/2$ is the angle that the vector makes with the positive y-axis. Hence the coordinates of a unit directional vector are called its *direction cosines*, since they are the cosines of the angle that the vector makes with the corresponding positive axes.

Before continuing, we should take a look at the trigonometric functions available in BASIC: SIN and COS, and the inverse function ATN. SIN and COS are functions with one parameter (an angle given in radians) and one result (a value between -1 and $+1$). The ATN function takes any value and calculates the angle in radians (in the so-called *principal range* between $-\pi/2$ and $+\pi/2$) whose tangent is that value.

This leads us to the problem of finding the angle that a general direction $q \equiv (x, y)$ makes with the positive x-axis, which is solved by routine 'angle' given in listing 3.4; 'angle' will be of great use in later chapters when we consider three-dimensional space.

Listing 3.4

```
8800 REM angle
8801 REM IN  : AX,AY
8802 REM OUT : THETA
8809 REM THETA is the angle made by line to (AX,AY) with +ve x-axis.
8810 IF ABS AX > 0.00001 THEN GO TO 8860
8819 REM line is vertical.
8820 LET THETA = PI/2
8830 IF AY < 0 THEN LET THETA = THETA + PI
8840 IF ABS AY < 0.00001 THEN LET THETA = 0
8850 RETURN
8859 REM line not vertical so it has finite tangent.
8860 LET THETA = ATN (AY/AX)
8870 IF AX < 0 THEN LET THETA = THETA + PI
8880 RETURN
```

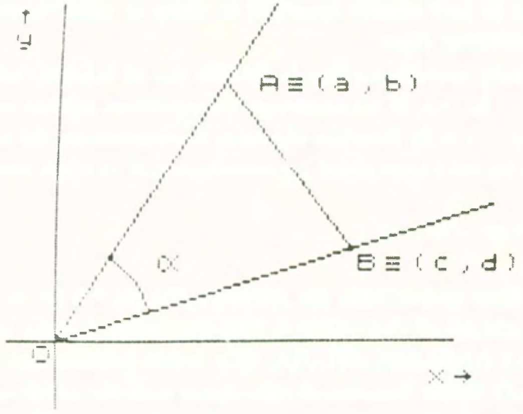


Figure 3.4

Now suppose we have two directional vectors (a, b) and (c, d); for simplicity we can assume that they are both unit vectors and they pass through the origin (see figure 3.4). We wish to calculate the acute angle, α, between these lines. From the figure we note that OA $= \sqrt{(a^2 + b^2)} = 1$ and OB $= \sqrt{(c^2 + d^2)} = 1$. So by the Cosine Rule

$$\mathrm{AB}^2 = \mathrm{OA}^2 + \mathrm{OB}^2 - 2\mathrm{OA} \times \mathrm{OB} \times \cos\alpha = 2 \times (1 - \cos\alpha)$$

But also by Pythagoras

$$\begin{aligned}\mathrm{AB}^2 &= (a-c)^2 + (b-d)^2 = (a^2 + b^2) + (c^2 + d^2) - 2\,(a \times c + b \times d)\\ &= 2 - 2\,(a \times c + b \times d)\end{aligned}$$

Thus $a \times c + b \times d = \cos\alpha$. It is possible that $a \times c + b \times d$ is negative, in which case $\cos^{-1}\,(a \times c + b \times d)$ is obtuse and the required acute angle is $\pi - \alpha$. Since $\cos(\pi - \alpha) = -\cos\alpha$, then the acute angle is given immediately by $\cos^{-1}\,(|a \times c + b \times d|)$. For example, given the two lines with direction cosines $(\sqrt{(3/2)}, 1/2)$ and $(-1/2, -\sqrt{(3/2)})$, we see that $a \times c + b \times d = -\sqrt{(3/2)}$ and thus $\alpha = \cos^{-1}(\sqrt{(3/2)}) = \pi/6$. This simple example was given in order to introduce the concept of a *scalar product* $\cdot$ of two vectors, $(a, b) \cdot (c, d) = a \times c + b \times d$. Scalar product is extendable into higher dimensional space (see chapter 7 for a three-dimensional example) and it always has the property that it gives the cosine of the angle between any pair of lines with directions defined by the two vectors.

Curves: Functional Representation versus Parametric Forms

A curve in two-dimensional space can be considered as a relationship between x and y coordinate values, the so-called *functional relationship*. Alternatively the coordinates can be individually specified in terms of other variables or parameters, the *parametric form*.

We have already seen that a line (a circular arc of infinite radius) may be expressed as $ay = bx + c$. If we rearrange the equation so that one side is zero (that is, $ay - bx - c = 0$) then the algebraic expression on the left-hand side of the equation is called a functional representation of the line and written

$$f(x, y) \equiv ay - bx - c$$

All, and only, those points with the property $f(x, y) = 0$ lie on the curve. This representation divides all the points in two-dimensional space into three sets, namely $f(x, y) = 0$ (the zero set), $f(x, y) > 0$ (the positive set) and $f(x, y) < 0$ (the negative set). If the function divides space into the curve and two other *connected areas* only (that is, any two points in a connected area can be joined by a curvilinear line that does not cross the curve), then these areas can be identified with the positive and negative sets defined by f. However, be wary, there are many elementary functions (for example, $g(x, y) = \cos(y) - \sin(x)$) that define not one but a series of curves and hence divide space into possibly an infinite number of connected areas (note $g(x, y) = g(x + 2m\pi, y + 2n\pi)$ for all integers m and n). So it is possible that two unconnected areas can both belong to the positive set.

Note that the functional representation need not be unique. We could have put the line in an equivalent form

$$f'(x, y) \equiv bx + c - ay$$

in which case the positive set of this function is the negative set of our original, and vice versa.

The case where the curve does divide space into two connected areas is very useful in computer graphics, as we shall see in a study of two-dimensional and (especially) three-dimensional graphics algorithms. For example, take the straight line

$$f(x, y) \equiv ay - bx - c$$

where a point (x_1, y_1) is on the same side of the line as (x_2, y_2) if and only if $f(x_1, y_1)$ has the same non-zero sign as $f(x_2, y_2)$. The functional representation tells us more about a point (x_1, y_1) than just which side of a line it lies – it also enables us to calculate the distance of the point from the line.

Suppose we have the above line, then its direction vector is (a, b). A line perpendicular to this will have direction vector $(-b, a)$ (why? the product of the tangents of two mutually perpendicular lines is -1: see McCrae, 1953). So the point $\boldsymbol{q}$ on the line closest to the point $\boldsymbol{p} \equiv (x_1, y_1)$ is of the form

$$q \equiv (x_1, y_1) + \mu(-b, a)$$

that is, a new line joining $\boldsymbol{p}$ to $\boldsymbol{q}$ is perpendicular to the original line. Since $\boldsymbol{q}$ lies on this original line

$$f(q) = f((x_1, y_1) + \mu(-b, a)) = 0$$

and hence

$$a \times (y_1 + \mu \times a) - b \times (x_1 - \mu \times b) - c = f(x_1, y_1) + \mu(a^2 + b^2) = 0$$

Hence $\mu = -f(x_1, y_1)/(a^2 + b^2)$. The point $\boldsymbol{q}$ is a distance $\mu \times |(-b, a)|$ from (x_1, y_1), which naturally means that the distance of (x_1, y_1) from the line is $\mu \times \sqrt{(a^2 + b^2)} = -f(x_1, y_1)/\sqrt{(a^2 + b^2)}$: the sign denotes on which side of the line the point is lying. If $a^2 + b^2 = 1$ then $|f(x_1, y_1)|$ gives the distance of the point (x_1, y_1) from the line.

This idea leads us straight to a way of implementing *convex areas*; that is, an area with the property that a straight line segment joining any two points within the area lies totally inside the area. We limit our study to convex polygons, however, since it is obvious that any convex area may be approximated by a polygon, providing it has enough sides.

Suppose we have a convex polygon with n vertices $\{\boldsymbol{p}_i \equiv (x_i, y_i) | i = 1, \ldots, n\}$ taken in order around the polygon either clockwise or anti-clockwise; we shall call such a description of a convex polygon an *oriented convex set* of vertices. The problem of finding whether such a set is clockwise or anti-clockwise is considered in chapter 7. The n boundary edges of the polygon are segments of the lines

$$f_i(x, y) \equiv (x_{i+1} - x_i) \times (y - y_i) - (y_{i+1} - y_i) \times (x - x_i)$$

where $i = 1, \ldots, n$, and the addition in the subscripts is modulo n (that is, $n + j \equiv j$ for $1 \leqslant j < n$). Try to explain why these formulae do actually describe the line segments!

This systematic definition of the lines enables us to define the inside of the convex area. Any given line segment, say the one joining $\boldsymbol{p}_i$ to $\boldsymbol{p}_{i+1}$ for some i, is such that the points inside the body must lie on the same side of this line as the remaining vertices of the polygon, in particular $\boldsymbol{p}_{i+2}$. So the inside is given by

$$\{(x, y) | \text{sign of } f_i(x, y) = \text{sign of } f_i(x_{i+2}, y_{i+2}) \neq 0: i = 1, \ldots, n\}$$

A point on the boundary is given by

$\{(x, y) |$ there exists one j, or two if (x, y) is a corner, where

$1 \leqslant j \leqslant n$ such that $f_j(x, y) = 0$ and

sign of $f_i(x, y)$ = sign of $f_i(x_{i+2}, y_{i+2}) \neq 0$: $i \neq j$ and $1 \leqslant i \leqslant n\}$

A point outside the area is defined by

$\{(x, y) \mid$ there exists one j, $1 \leqslant j \leqslant n$ such that

$0 \neq$ sign of $f_j(x, y) \neq$ sign of $f_j(x_{j+2}, y_{j+2}) \neq 0\}$

Naturally the additions of subscripts are all modulo n.

Example 3.4
Suppose we are given the convex polygon with vertices (1, 0), (5, 2), (4, 4) and (−2, 1) (see figure 3.5). In this order the vertices obviously have an anti-clockwise orientation. Are the points (3, 2), (1, 4), (3, 1) inside, outside or on the boundary of the polygon? What is the distance of (4, 4) from the first line?

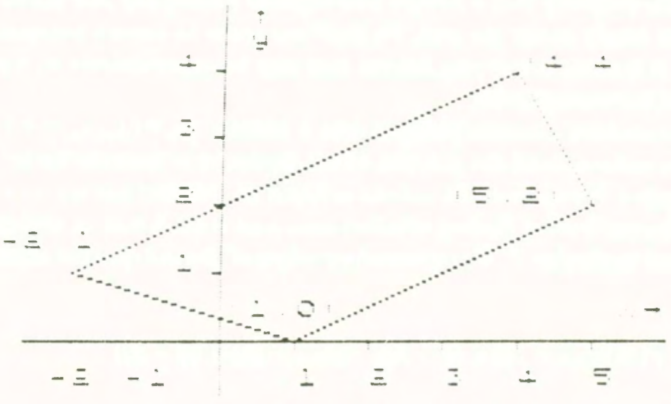


Figure 3.5

$$f_1(x, y) \equiv (5 - 1) \times (y - 0) - (2 - 0) \times (x - 1) \equiv 4y - 2x + 2$$

$$f_2(x, y) \equiv (4 - 5) \times (y - 2) - (4 - 2) \times (x - 5) \equiv -y - 2x + 12$$

$$f_3(x, y) \equiv (-2 - 4) \times (y - 4) - (1 - 4) \times (x - 4) \equiv -6y + 3x + 12$$

$$f_4(x, y) \equiv (1 + 2) \times (y - 1) - (0 - 1) \times (x + 2) \equiv 3y + x - 1$$

Hence point (3, 2) is inside the body because $f_1(3, 2) = 4$ and $f_1(4, 4) = 10$; $f_2(3, 2) = 4$ and $f_2(-2, 1) = 15$; $f_3(3, 2) = 9$ and $f_3(1, 0) = 15$; $f_4(3, 2) = 8$ and $f_4(5, 2) = 10$ – all with the same positive signs.
Point (1, 4) is outside the body because $f_3(1, 4) = -9$ and $f_3(1, 0) = 15$ – opposite signs.

Point (3, 1) is on the boundary because $f_1\ (3, 1) = 0, f_2\ (3, 1) = 5, f_3\ (3, 1) = 15$ and $f_4\ (3, 1) = 5$.

In fact there is no need to work out $f_i(x_{i+2}, y_{i+2})$ for every i, they all have the same sign so once we have calculated $f_1\ (x_3, y_3)$ then we can work with this value throughout.

(4, 4) is a distance $f_1\ (4, 4)/\sqrt{(4^2 + 2^2)} = 10/\sqrt{20} = \sqrt{5}$.

Exercise 3.5
Imagine two convex polygons that intersect one another. The area of intersection is also a convex polygon. Use the methods mentioned in this chapter to calculate the vertices of the new polygon.

Having dealt with the functional representation of a line, what about the parametric form? We noted that this form is one where the x-coordinate and y-coordinate of a general point on the curve are given in terms of parameter(s) (which might be the x-value and the y-value themselves), together with a range for the parameter. So we have already seen a parametric form of a line: it is simply the base and directional representation

$$\begin{aligned} b + \mu d &\equiv (x_1, y_1) + \mu(x_2, y_2) \\ &\equiv (x_1 + \mu \times x_2, y_1 + \mu \times y_2) \quad \text{where} \quad -\infty < \mu < \infty \end{aligned}$$

μ is the parameter, and $x_1 + \mu \times x_2$ and $y_1 + \mu \times y_2$ are the respective x-value and y-value, depending only on variable μ.

We can also produce functional representations and parametric forms for most well-behaved curves. For example, a sine curve is given by $f(x, y) \equiv y - \sin(x)$ in functional representation, and by $(x, \sin(x))$ with $-\infty < x < \infty$ in its parametric form. The general conic section (ellipse, parabola and hyperbola) is represented by the general function

$$f(x, y) \equiv a \times x^2 + b \times y^2 + h \times x \times y + f \times x + g \times y + c$$

where coefficients a, b, c, f, g, h uniquely identify a curve. A circle centred at the origin of radius r has $a = b = 1, f = g = h = 0$ and $c = -r^2$, whence $f(x,y) \equiv x^2 + y^2 - r^2$. All the points (x, y) on the circle are such that $f(x, y) = 0$, the inside of the circle has $f(x, y) < 0$, and the outside of the circle $f(x, y) > 0$. The parametric form of this circle is $(r \cos \alpha, r \sin \alpha)$ where $0 \leqslant \alpha \leqslant 2\pi$. (We have already met the parametric form of a circle, ellipse and spiral in chapter 2).

It is very useful to experiment with these (and other) concepts in two-dimensional geometry. There will be many occasions when it is necessary to include these ideas in programs, as well as the ever-present need when generating coordinate data for diagrams.

Example 3.5
Suppose we wish to draw a circular ball (radius r) disappearing down an elliptical hole (major axis a, minor axis b), see figure 3.6. Parts of both the ellipse and circle are obscured.

Let the ellipse be centred on the origin with the major axis horizontal, and the centre of the circle a distance d vertically above the origin. The ellipse has functional representation

$$f_e(x, y) \equiv x^2/a^2 + y^2/b^2 - 1$$

and in parametric form

$$(a \times \cos\alpha, b \times \sin\alpha) \text{ with } 0 \leqslant \alpha \leqslant 2\pi$$

For the circle

$$f_c(x, y) \equiv x^2 + (y - d)^2 - r^2$$

and in parametric form

$$(r \times \cos\lambda, d + r \times \sin\lambda) \quad \text{where} \quad 0 \leqslant \lambda \leqslant 2\pi$$

To generate the picture we must find the points (x, y) common to the circle and ellipse (if any). As a useful demonstration we shall mix the representations in searching for a solution, using the functional representation for the circle and the parametric form of the ellipse.

So we are searching for the points $(x, y) \equiv (a \times \cos\alpha, b \times \sin\alpha)$ on the ellipse, which also satisfy $f_c(x, y) = 0$. That is

$$a^2 \times \cos^2\alpha + (b \times \sin\alpha - d)^2 - r^2 = 0$$

and $a^2 \times \cos^2\alpha + b^2 \times \sin^2\alpha - 2 \times b \times d \times \sin\alpha + d^2 - r^2 = 0$

And since $\cos^2\alpha = 1 - \sin^2\alpha$

$$(b^2 - a^2) \times \sin^2\alpha - 2 \times b \times d \times \sin\alpha + a^2 + d^2 - r^2 = 0$$

This is a simple quadratic equation in the unknown $\sin\alpha$, which is easily solved (the quadratic equation $Ax^2 + Bx + C = 0$ has two roots $(-B \pm \sqrt{(B^2 - 4 \times A \times C)})/(2 \times A)$). For each value of $\sin\alpha$ we can find values for α with $0 \leqslant \alpha \leqslant 2\pi$ (if they exist) and we can then calculate the points of intersection $(a \times \cos\alpha, b \times \sin\alpha)$.

There is no hard and fast rule regarding which representation to use in any given situation – a *feel* for the method is required and that comes only with experience.

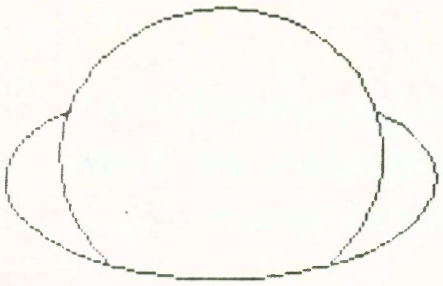


Figure 3.6

Exercise 3.6
Write a program that will draw figure 3.6.

Complete Programs

I. 'lib1' and listing 3.1: no data required.
II. Listings 2.1, 2.8 and 3.2: data are two coordinate pairs (X1, Y1) and (X2, Y2), where $-3 < X1, X2 < 3$ and $-2.1 < Y1, Y2 < 2.1$.
Note: from this point listing 3.3 ('clip' and a new version of 'lineto') will replace listing 2.5 in 'lib1'.
III. The same as I above, but with the new 'lib1': change HORIZ to 1.5 and VERT to 1.

4 *Matrix Representation of Transformations on Two-Dimensional Space*

In chapter 2 we saw the need to translate pictures of objects about the screen. Rather than perpetually change the screen coordinate system, it is conceptually much easier to define an object in the most simple terms possible (as vertices in the form of pixel or coordinate values, together with line and area information related to the vertices), and then transform the object to various parts of the screen while keeping the screen coordinate system fixed. We shall restrict ourselves to linear transformations (see below). It will often be necessary to transform a large number of vertices, and to do this efficiently we use *matrices*. Before looking at such matrix representations we should explain exactly what is meant by a matrix, and also by a *column vector*. In fact we restrict ourselves to square matrices; to 3 × 3 (said 3 by 3) for the study of two-dimensional space, and later we shall use 4 × 4 matrices when considering three-dimensional space. Such a 3 × 3 matrix (A say) is simply a group of real numbers placed in a block of 3 rows by 3 columns: a column vector (D say) is a group of numbers placed in a column of 3 rows

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}$$

A general entry in the matrix is usually written A_{ij}, the first subscript denotes the i^{th} row, and the second subscript the j^{th} column (for example, A_{23} represents the value in the second row, third column). The entry in the column vector, D_i, denotes the value in the i^{th} row. All these named entries will be explicitly replaced by numerical values and it is important to realise that the *information* stored in a matrix or column vector is not just the individual values but also the position of these values within the matrix or vector. Naturally BASIC programs are written along a line (no subscripts or superscripts), and hence matrices and vectors are implemented as arrays and the subscript values appear inside round brackets following the array identifier.

Matrices can be added. Matrix $C = A + B$, the sum of two matrices A and B, is defined by the general entry C_{ij} thus

$$C_{ij} = A_{ij} + B_{ij} \quad 1 \leqslant i, j \leqslant 3$$

Matrix A can be multiplied by a scalar k to form a matrix B

$$B_{ij} = k \times A_{ij} \quad 1 \leqslant i, j \leqslant 3$$

We can multiply a matrix A by a column vector D to produce another column vector E thus

$$E_i = A_{i1} \times D_1 + A_{i2} \times D_2 + A_{i3} \times D_3 = \sum_k A_{ik} \times D_k \quad \text{where } 1 \leqslant i \leqslant 3$$

The i^{th} row element of the new column vector is the sum of the products of the corresponding elements of the i^{th} row of the matrix with those in the column vector.

Furthermore, we can calculate the product (matrix) $C = A \times B$ of two matrices A and B

$$C_{ij} = A_{i1} \times B_{1j} + A_{i2} \times B_{2j} + A_{i3} \times B_{3j} = \sum_k A_{ik} \times B_{kj} \quad \text{where } 1 \leqslant i, j \leqslant 3$$

We take the sum (in order) of the elements in the i^{th} row of the first matrix multiplied by the elements in the j^{th} column of the second. It should be noted that the product of matrices is not necessarily *commutative*; that is, $A \times B$ need not be the same as $B \times A$. For example

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ but } \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Experiment with these ideas until you have enough confidence to use them in the theory that follows. For those who want more details about the theory of matrices we recommend books by Finkbeiner (1978) and by Stroud (1982).

There is a special matrix called the *identity matrix I* (sometimes called the unit matrix)

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Also for every matrix A we can calculate its *determinant* det (A)

$$\det(A) = A_{11} \times (A_{22} \times A_{33} - A_{23} \times A_{32}) + A_{12} \times (A_{23} \times A_{31} - A_{21} \times A_{33}) \\ + A_{13} \times (A_{21} \times A_{32} - A_{22} \times A_{31})$$

Any matrix whose determinant is non-zero is called *non-singular*, and with zero determinant *singular*. All non-singular matrices A have an *inverse* A^{-1}, which has the property that $A \times A^{-1} = I$ and $A^{-1} \times A = I$. For methods of calculating an inverse of a matrix see Finkbeiner (1978): we give a listing in chapter 7 (listing 7.5), which uses the Adjoint method.

We now consider transforming points in space. Suppose a point (x, y) – 'before' – is transformed to (x', y') – 'after'. We understand the transformation completely if we can give equations relating the 'before' and 'after' points. A linear transformation is one that defines the 'after' point in terms of linear combinations of the coordinates of the 'before' point; that is, the equations contain only multiples of x, y and additional real values -- it includes neither non-unit powers or multiples of x and y, nor other variables. Such equations may be written

$$x' = A_{11} \times x + A_{12} \times y + A_{13}$$
$$y' = A_{21} \times x + A_{22} \times y + A_{23}$$

The A values are called the *coefficients* of the equation. As we can see, the result of the transformation is a combination of multiples of x-values, y-values and unity. We may add another equation

$$1 = A_{31} \times x + A_{32} \times y + A_{33}$$

For this to be true for all values of x and y, we see that $A_{31} = A_{32} = 0$ and $A_{33} = 1$. This may seem a pointless exercise but we shall see that it is very useful. For if we set each point vector (x, y) (also called a *row vector* for obvious reasons) in the form of a three-dimensional column vector

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

then the above three equations can be written in the form of a matrix multiplying a column vector

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{13} & A_{13} \\ A_{21} & A_{23} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

So if we store the transformation as a matrix, we can transform every required point by considering it as a column vector and premultiplying it by the matrix.

Many writers of books on computer graphics do not like the use of column vectors. They prefer to extend the row vector (for example, (x, y) to $(x, y, 1)$), and postmultiply the row vector by the matrix so that the above equations in matrix form become

$$(x', y', 1) = (x, y, 1) \times \begin{pmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \\ A_{13} & A_{23} & A_{33} \end{pmatrix}$$

Note that this matrix is the *transpose* of the matrix of coefficients in the equations. This causes a great deal of confusion among those who are not confident in the use of matrices. It is for this reason that in this book we keep to the column vector notation. As you get more practice in the use of matrices it is a good idea to rewrite some (or all) of the following transformation routines in the other notation. It is not important which method you use finally, *as long as you are consistent*. (Note the transpose B of a matrix A is given by $B_{ij} = A_{ji}$, where $1 \leqslant i, j \leqslant 3$.)

Combination of Transformations

A very useful property of this matrix representation of transformations is that if we wish to combine two transformations on (say) transformation (= matrix) A followed by transformation B, then the combined transformation is represented by their product $C = B \times A$: note the order of multiplication – the matrix representing the first transformation is *premultiplied* by the second. This is because the final matrix will be used to premultiply a column vector representing a point, and so the first transformation matrix must appear on the right of the product and the last on the left. (If we had used the row vector method then the product would appear in the *natural order* from left to right – this is the price we pay for identifying the transformation matrix with the coefficients of the equation.)

So we need to introduce a routine 'mult2', which forms the product of two matrices. The BASIC computer language does not allow the transmission of array parameters into routines, so we must invent an efficient means of coping with this limitation. We assume that all matrix multiplication operates on matrices A and R giving the product matrix B, and after the product is complete B is copied back into R. The reason for the choice of identifiers and the final copy will become evident as we progress. We also need a routine ('idR2'), which sets R to the identity matrix. Should we need to form the product of a sequence of matrices we first set $R = I$ and then for each of the matrices from right to left, we name each A and call the routine 'mult2' in turn. At the end of the process, R contains the matrix product of the sequence (see listing 4.1).

Listing 4.1

```
9100 REM mult2
9101 REM IN  : A(3,3),R(3,3)
9102 REM OUT : R(3,3)
9110 FOR I = 1 TO 3
9120 FOR J = 1 TO 3
9130 LET AR = 0
9140 FOR K = 1 TO 3
9150 LET AR = AR + A(I,K)*R(K,J)
9160 NEXT K
9170 LET B(I,J) = AR
9180 NEXT J
9190 NEXT I
9200 FOR I = 1 TO 3
9210 FOR J = 1 TO 3
9220 LET R(I,J) = B(I,J)
9230 NEXT J
9240 NEXT I
9250 RETURN

9300 REM idR2
9302 REM OUT : R(3,3)
9310 FOR I = 1 TO 3
9320 FOR J = 1 TO 3
9330 LET R(I,J) = 0
9340 NEXT J
9350 LET R(I,I) = 1
9360 NEXT I
9370 RETURN
```

All natural transformations may be reduced to a combination of three basic forms of linear transformation: translation, scaling and rotation about the co-ordinate origin. It should also be noted that all valid applications of these transformations return non-singular matrices. The routines that follow generate a matrix called A for each of the three types of transformation, so that each transformation routine can be used in conjunction with 'mult2' to produce combinations of transformations.

Translation

A 'before' point (x, y) is moved by a vector (TX, TY) to (x', y') say. This produces the equations

$$x' = 1 \times x + 0 \times y + \mathrm{TX}$$

$$y' = 0 \times x + 1 \times y + \mathrm{TY}$$

so the matrix describing this transformation is

$$\begin{pmatrix} 1 & 0 & \mathrm{TX} \\ 0 & 1 & \mathrm{TY} \\ 0 & 0 & 1 \end{pmatrix}$$

And a routine, 'tran2', for generating such a matrix A, given the values TX and TY is given in listing 4.2.

Listing 4.2

```
9000 REM tran2
9001 REM IN  : TX,TY
9002 REM OUT : A(3,3)
9010 FOR I = 1 TO 3
9020 FOR J = 1 TO 3
9030 LET A(I,J) = 0
9040 NEXT J
9050 LET A(I,I) = 1
9060 NEXT I
9070 LET A(1,3) = TX: LET A(2,3) = TY
9080 RETURN
```

Scaling

The x-coordinate of a point in space is scaled by a factor SX, and the y-coordinate by SY, thus

$$x' = \mathrm{SX} \times x + 0 \times y + 0$$

$$y' = 0 \times x + \mathrm{SY} \times y + 0$$

giving the matrix

$$\begin{pmatrix} \mathrm{SX} & 0 & 0 \\ 0 & \mathrm{SY} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Usually SX and SY are both positive, but if one or both are negative this creates a reflection as well as a scaling. In particular, if SX = −1 and SY = 1 then the point is reflected about the y-axis. A program segment, 'scale2', to produce such a scaling matrix A given SX and SY is given in listing 4.3.

Listing 4.3

```
8900 REM scale2
8901 REM IN  : SX,SY
8902 REM OUT : A(3,3)
8910 FOR I = 1 TO 3
8920 FOR J = 1 TO 3
8930 LET A(I,J) = 0
8940 NEXT J
8950 NEXT I
8960 LET A(1,1) = SX: LET A(2,2) = SY
8970 LET A(3,3) = 1
8980 RETURN
```

Rotation about the Origin

If we rotate a point in an anti-clockwise direction (the normal mathematical orientation) about the origin by an angle θ then the equations are

$$x' = \cos\theta \times x - \sin\theta \times y + 0$$

$$y' = \sin\theta \times x + \cos\theta \times y + 0$$

and the matrix is

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The routine, 'rot2', to produce a rotation matrix, A, for an angle θ is given in listing 4.4.

Listing 4.4

```
8600 REM rot2
8601 REM IN  : THETA
8602 REM OUT : A(3,3)
8610 FOR I = 1 TO 3
8620 FOR J = 1 TO 3
8630 LET A(I,J) = 0
8640 NEXT J
8650 NEXT I
8660 LET A(3,3) = 1
8670 LET CT = COS THETA: LET ST = SIN THETA
8680 LET A(1,1) = CT: LET A(2,2) = CT
8690 LET A(1,2) = -ST: LET A(2,1) = ST
8700 RETURN
```

Inverse Transformations

For every transformation there is an inverse transformation that will restore the points in space to their original position. If a transformation is represented by a matrix A, then the inverse transformation is represented by the inverse matrix A^{-1}. There is no need to calculate this inverse using listing 7.5, we can find it directly by using listings 4.2, 4.3 and 4.4, with parameters derived from the parameters of the original transformation

(1) a translation by (TX, TY) is inverted by a translation by (–TX, –TY);
(2) a scaling by SX and SY is inverted by a scaling by 1/SX and 1/SY (naturally both SX and SY are non-zero, for otherwise the two-dimensional space would contract into a line or a point);
(3) a rotation by an angle θ is inverted by a rotation by an angle $-\theta$;

(4) if the transformation matrix is a product of a number of translation, scaling and rotation matrices $A \times B \times C \times \ldots \times L \times M \times N$ (say), then the inverse transformation matrix is

$$N^{-1} \times M^{-1} \times L^{-1} \times \ldots \times C^{-1} \times B^{-1} \times A^{-1}$$

Note the order of multiplication!

The Placing of an Object

We are often required to draw a given object at various points on the screen, and at arbitrary orientations. It would be very inefficient to calculate by hand the coordinates of vertices for each position of the object and input them to the program. Instead we define first an arbitrary but fixed coordinate system for two-dimensional space, which we call the ABSOLUTE system. Then we give the coordinates of the vertices of the object in some simple way, usually about the origin, which we call the SETUP position. Lines and areas within the object are defined in terms of the vertices. We can then use matrices to move the vertices of the object from the SETUP to the ACTUAL position in the ABSOLUTE system. The lines and areas maintain their relationship with the now transformed vertices. The matrix that relates the SETUP to ACTUAL position will be called P throughout this book (we sometimes give it a letter subscript to identify it uniquely from other such matrices). Because of the restriction of not passing arrays as parameters into subprograms, we shall not normally explicitly generate array P, instead it will be implicitly used to update the array R.

Looking at the Object

Thus objects in a scene can be moved relative to the ABSOLUTE coordinate axes. When observing such a scene, the eye is assumed to be looking directly at point (DX, DY) of the ABSOLUTE system and the head tilted through an angle α. It would be convenient to assume that it is looking at the origin and there is no tilt of the head (we call this the OBSERVED position). Therefore we generate another matrix that will transform space so that the eye is moved from its ACTUAL position to this OBSERVED position. The ACTUAL to OBSERVED matrix is named Q throughout this book, and is achieved by first translating all points in space by a vector (–DX, –DY), matrix A, and then rotating them by an angle $-\alpha$, matrix B (note the minus signs!). Thus $Q = B \times A$, which is generated in routine 'look2', listing 4.5. Normally we do not calculate Q explicitly, as usually it is used only to update R; however, if it is necessary to use the values of the matrix repeatedly then obviously it is sensible to store Q.

Listing 4.5

```
8200 REM look2/general
8202 REM OUT : R(3,3)
8210 INPUT "(DX,DY) ";DX;",";DY
8220 INPUT "ALPHA ";ALPHA
8229 REM look at (DX,DY).
8230 LET TX = -DX: LET TY = -DY
8240 GO SUB tran2: GO SUB mult2
8249 REM tilt head through ALPHA radians.
8250 LET THETA = -ALPHA
8260 GO SUB rot2: GO SUB mult2
8270 RETURN
```

Drawing an Object

Combining the SETUP to ACTUAL matrix P, with the ACTUAL to OBSERVED matrix Q, we get the SETUP to OBSERVED matrix $R = Q \times P$ (we shall always use R to denote this matrix: and remember R is always the result of our 'mult2' routine). Transforming all the SETUP vertices by R, with the corresponding movement of line and area information, means that the coordinates of the object are given relative to the observer who is looking at the origin of the ABSOLUTE coordinate system with head upright, and who is in fact really looking at a graphics screen. So we identify the ABSOLUTE coordinate system with the system of the screen to find the position of the vertices on the screen, and then draw the vertices, lines and areas that compose the object. In practice this is achieved by a *construction routine* that uses matrix R. It will set up the vertex, line and area information, transform the vertices using R, and perhaps finally draw the object (see example 4.1 below). Later we shall see that there are certain situations where it is more efficient to store the vertex, line and area information. For example, the vertex coordinates can be stored in arrays X and Y, and line information in a two-dimensional array L. Vertices can be stored in their SETUP, ACTUAL or OBSERVED position – it really depends on the context of the program. This SETUP to ACTUAL to OBSERVED method will enable us to draw a dynamic series of scenes – objects can move relative to the ABSOLUTE axes, and to themselves, while simultaneously the observer can move independently around the scene. To start with, however, we consider the simplest case of a fixed scene.

Complicated Pictures – the 'Building Brick' Method

We can draw pictures that contain a number of similar objects. There is no need to produce a new routine for each occurrence of the object, all we do each time is calculate a new SETUP to OBSERVED matrix and enter this into the same routine. Naturally we shall require one routine for each new type of object in the picture. The final picture is achieved by the execution of a routine we name 'scene2', which will be called from the standard main program (listing 4.6). This main program simply defines the labels of the various subprograms, declares arrays, centres the graphics area having INPUT HORIZ and VERT, and finally calls 'scene2'.

Listing 4.6

```
100 REM main program
109 REM define identifiers for initialisation and 2-D plot routines.
110 LET start = 9700: LET setorigin = 9600: LET moveto = 9500
    : LET lineto = 9400: LET clip = 8400
120 LET rot2 = 8600: LET angle = 8800: LET scale2 = 8900: LET tran2 = 9000
    : LET mult2 = 9100: LET idR2 = 9300
130 LET scene2 = 6000: LET look2 = 8200
139 REM initialise and centre graphics area.
140 INPUT "HORIZ ",HORIZ,"VERT ",VERT
150 GO SUB start
160 LET XMOVE = HORIZ*0.5: LET YMOVE = VERT*0.5
170 GO SUB setorigin
139 REM set the scene.
180 GO SUB scene2
190 STOP
```

'scene2' will first call 'look2' and generate Q, and if more than one object is to be drawn then we store it. For each individual object (or *brick*) we calculate a matrix P and call the required construction routine using $R = Q \times P$. All the bricks finally build into the finished picture. To distinguish between different occurrences of these matrices in what follows, we sometimes add a subscript to the names P and R.

This modular approach to solve the problem of defining and drawing a picture may not be the most efficient, but from our experience it does greatly clarify the situation for beginners, enabling them to ask the right questions about constructing a required scene. Also when dealing with animation we shall see that this approach will minimise problems in scenes where not only are the objects moving relative to one another, but also the observer himself is moving. Naturally if the head is upright then matrix Q can be replaced by a call to 'setorigin', which changes the screen coordinate system. Or if the eye is looking at the origin, head upright, then Q is the identity matrix I; hence it plays no part in transforming the picture and the 'look2' routine may be ignored. We shall make no such assumptions and work with the most general situation: it is a useful exercise throughout this book for the reader to *cannibalise* our programs in order to make them efficient for specific cases. It is our aim to explain these concepts in the most general and straightforward terms, even at the expense of efficiency and speed. The reader can return to these programs when he is ready and fully understands these ideas of transforming space. Later we shall give some hints on how to make these changes, but at the moment this would only confuse the issue.

However, the most important reason for this modular approach will be seen when we come to draw pictures of three-dimensional objects. We shall define these three-dimensional constructions as an extension of the ideas above, and full understanding of two-dimensional transformations is essential before we go on to higher dimensions.

Example 4.1

Consider a simple *space ship* SETUP pointing in the positive x-direction (that is, making an angle 0 radians with the x-direction). The ship is defined by five line segments joining, in order, the points (3, 0), (0, 0), (–1, 1), (2, 0), (–1, –1) and back to (0, 0). See figure 4.1; which is a ship drawn on a screen 5 units by 3 units, where the SETUP to ACTUAL matrix is the identity and the ACTUAL to OBSERVED matrix is such that the observer is looking at the point (1, 0) with head upright. Listing 4.7 gives the necessary routine 'scene2' that moves the object into position, and listing 4.8 is the required construction routine 'ship'. Note that 'ship', which uses matrix R to transform the vertices (and hence the object) into their OBSERVED position, does not store the vertex values for this position in a permanent data-base. Instead the values are kept in arrays X and Y for the duration of the routine, and if the routine is re-entered to draw another space ship then these array locations are used again.

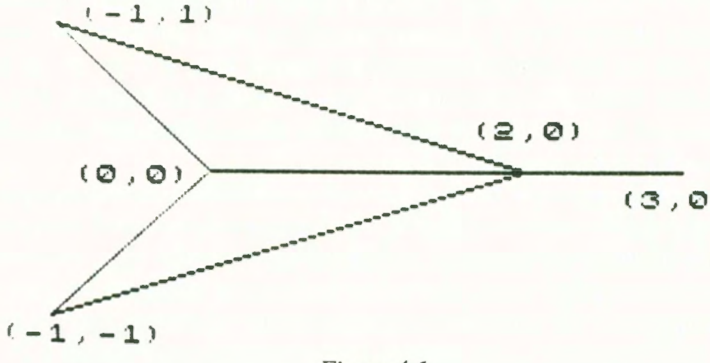


Figure 4.1

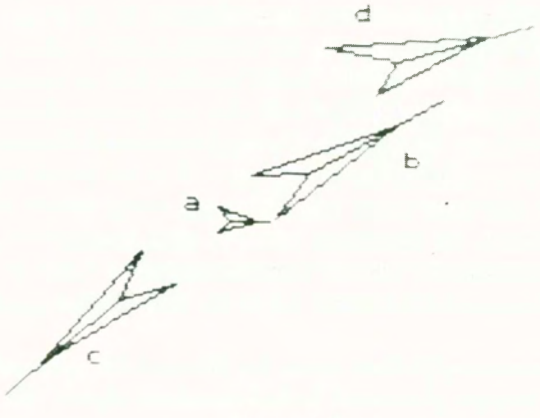


Figure 4.2

Listing 4.7

```
6000 REM scene2/look2 ; ship(not stored)
6010 DIM X(6): DIM Y(6)
6020 DIM A(3,3): DIM B(3,3): DIM R(3,3)
6030 LET ship = 6500
6039 REM place the observer.
6040 GO SUB idR2: GO SUB look2
6049 REM define and draw object.
6050 GO SUB ship
6060 RETURN
```

Listing 4.8

```
6500 REM ship/ not stored
6509 REM IN : R(3,3)
6510 DATA 3,0,0,0,-1,1,2,0,-1,-1,0,0
6520 RESTORE ship
6530 FOR I = 1 TO 6
6539 REM read coordinates of SETUP object.
6540 READ XX,YY
6549 REM move object into OBSERVED position.
6550 LET X(I) = XX*R(1,1) + YY*R(1,2) + R(1,3)
6560 LET Y(I) = XX*R(2,1) + YY*R(2,2) + R(2,3)
6570 NEXT I
6579 REM join vertices in order.
6580 LET XPT = X(1): LET YPT = Y(1): GO SUB moveto
6590 FOR I = 2 TO 6
6600 LET XPT = X(I): LET YPT = Y(I): GO SUB lineto
6610 NEXT I
6620 RETURN
```

Example 4.2

Suppose we wish to draw figure 4.2, which includes four space ships labelled (a), (b), (c) and (d) on a screen 60 units by 40 units. For simplicity in this picture we assume Q is the identity matrix, so the head is upright and the eye looks at the SETUP origin. Ship (a) is placed identically to its SETUP position; that is, $R_a = I$, whereas ship (b) is moved from its SETUP to ACTUAL position by the following transformations

(1) scale the figure with SX = 4 and SY = 2, producing matrix A;
(2) rotate the figure through $\pi/6$ radians, matrix B;
(3) translate figure by TX = 6 and TY = 4, matrix C

$$A = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} \sqrt{3}/2 & -1/2 & 0 \\ 1/2 & \sqrt{3}/2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 0 & 6 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{pmatrix}$$

The complete transformation is given by $R_b = Q \times P_b = I \times P_b = P_b = C \times B \times A$ (Note the order of matrix multiplication, and that the subscript distinguishes the placing of ship (b) from the others).

If instead we used the order $A \times B \times C$ (giving matrix P_d), then

$$P_b = \begin{pmatrix} 2\sqrt{3} & -1 & 6 \\ 2 & \sqrt{3} & 4 \\ 0 & 0 & 1 \end{pmatrix} \quad P_d = \begin{pmatrix} 2\sqrt{3} & -2 & 12\sqrt{3}-8 \\ 1 & \sqrt{3} & 4\sqrt{3}+6 \\ 0 & 0 & 1 \end{pmatrix}$$

which are two obviously different transformations. Matrix $R_d = Q \times P_d = I \times P_d$ produces ship (d). Note how this ship is assymetrical. Be very careful with the use of the scaling transformation – remember scaling is defined about the origin and this will cause distortions in the shape of an object that is moved away from the origin!

To illustrate this example further we show how to calculate the ACTUAL position of ship (b) on the screen by setting the coordinates in the form of a column vector and premultiplying them by matrix $R_b = I \times P_b$; for example

$$\begin{pmatrix} 2\sqrt{3} & -1 & 6 \\ 2 & \sqrt{3} & 4 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 6\sqrt{3}+6 \\ 10 \\ 1 \end{pmatrix} \quad \text{etc.}$$

When returned to normal vector form we see that the five vertices have been transformed to $(6\sqrt{3} + 6, 10)$, $(6, 4)$, $(5 - 2\sqrt{3}, \sqrt{3} + 2)$, $(4\sqrt{3} + 6, 8)$ and $(7 - 2\sqrt{3}, 2 - \sqrt{3})$ respectively.

Ship (c) is ship (b) reflected in the line $3y = -4x - 9$. This line cuts the y-axis at $(0, -3)$ and makes an angle $\alpha = \cos^{-1}(-3/5) = \sin^{-1}(4/5) = \tan^{-1}(-3/4)$ with the positive x-axis. If we move space by a vector $(0, 3)$, matrix D say, this line will go through the origin. Furthermore, if we rotate space by $-\alpha$, matrix E say, the line now is identical with the x-axis. Matrix F can reflect the ship in the x-axis, E^{-1} puts the line back at an angle α with the x-axis, and finally D^{-1} returns the line to its original position. Matrix $G = D^{-1} \times E^{-1} \times F \times E \times D$ will therefore reflect all the ACTUAL vertices of ship (a) about the line $3y = -4x - 9$, and $R_c = I \times P_c = G \times P_b$ can therefore be used to draw ship (c). That is, we use matrix P_b to move the ship to position (b) and then G to place it in position (c).

$$D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{pmatrix} \quad E = \begin{pmatrix} -3/5 & 4/5 & 0 \\ -4/5 & -3/5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad F = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and

$$R_c = \frac{1}{25}\begin{pmatrix} -48 - 14\sqrt{3} & 7 - 24\sqrt{3} & -210 \\ 14 - 48\sqrt{3} & 24 + 7\sqrt{3} & -170 \\ 0 & 0 & 25 \end{pmatrix}$$

Figure 4.2 is drawn using the new 'scene2' routine of listing 4.9: note that this 'scene2' does not call 'look2', since it is assumed that the eye is looking at the origin with the head erect. The main program and the 'ship' routine, as well as all the other graphics package routines, stay unchanged.

Listing 4.9

```
6000 REM scene2/ no look2 ; 4 ships (not stored)
6010 DIM X(6) : DIM Y(6)
6020 DIM A(3,3): DIM B(3,3): DIM R(3,3)
6030 LET ship = 6500
6039 REM OBSERVED=ACTUAL, no need to call 'look2'.
6034 REM ship a).
6040 GO SUB idR2: GO SUB ship
6049 REM ship b).
6050 LET SX = 4: LET SY = 2
6060 GO SUB scale2: GO SUB mult2
6070 LET THETA = PI/6
6080 GO SUB rot2: GO SUB mult2
6090 LET TX = 6: LET TY = 4
6100 GO SUB tran2: GO SUB mult2
6110 GO SUB ship
6119 REM ship c).
6120 LET AX = -3: LET AY = 4
6130 GO SUB angle
6140 LET TX = 0: LET TY = 3
6150 GO SUB tran2: GO SUB mult2
6160 LET THETA = -THETA
6170 GO SUB rot2: GO SUB mult2
6180 LET SX = 1: LET SY = -1
6190 GO SUB scale2: GO SUB mult2
6200 LET THETA = -THETA
6210 GO SUB rot2: GO SUB mult2
6220 LET TX = 0: LET TY = -3
6230 GO SUB tran2: GO SUB mult2
6240 GO SUB ship
6249 REM ship d).
6250 GO SUB idR2
6260 LET TX = 6: LET TY = 4
6270 GO SUB tran2: GO SUB mult2
6280 LET THETA = PI/6
6290 GO SUB rot2: GO SUB mult2
6300 LET SX = 4: LET SY = 2
6310 GO SUB scale2: GO SUB mult2
6320 GO SUB ship
6330 RETURN
```

Exercise 4.1

In order to convince yourself that this program may be used to deal with the general situation, you should run this program using non-zero values of DX, DY or α so that the ACTUAL to OBSERVED matrix Q is not the identity matrix. Your 'scene2' routine should call 'look2' to calculate Q, which must be stored. Then for each object in the scene, in turn, calculate the SETUP to ACTUAL matrix P (which 'mult2' places in R), premultiply it by Q (which has to be copied into matrix A for use with 'mult2') and finally enter the construction routine with the product matrix $R = Q \times P$. Make sure that the 'lineto' routine contains the 'clip' option or you will find your program fails when it tries to draw outside the rectangle of the graphics area.

Exercise 4.2
Use the above routines to draw diagrams that are similar to figure 4.2, but where the number, position and directions of the ships are read in from the keyboard. You can produce routines to draw more complicated objects; we chose a very simple example so that the algorithms would not be obscured by the complexity of objects. The above method can deal with as many vertices and lines as the Spectrum can handle within time and storage limitations. The objects need not be line drawings only, you can draw coloured areas (polygons bounded by transformed vertices).

Exercise 4.3
Using loops in the program we can draw ordered sequences of the objects; for example, they can all have the same orientation but with points of reference (the origin in the SETUP position) equally spaced along any line $\boldsymbol{p} + \mu\boldsymbol{q}$. We can set up a loop with index parameter μ and draw one ship for each pass through the loop. For each value of μ we can alter the parameters of translation in a regular way within the loop (using $\mu, \boldsymbol{p}$ and $\boldsymbol{q}$). The new values of these parameters are used to calculate a different SETUP to ACTUAL matrix for each occurrence, and this moves the object into a new ACTUAL position. $R = Q \times P = I \times P$ is used to observe and draw each object on the screen. With these ideas, construct a set of *battle formations* of the above type of ship on the screen.

Efficient Use of Matrices

It is obvious that whatever combination of transformations we use, the third row of every matrix will always be (0 0 1). If we work with the top two rows of the matrix only, it will make our routine much more efficient. We still keep 3 X 3 matrices rather than 2 X 3 (which is really all we need), because we may have previously written other routines that assume 3 X 3 matrices. ReDIMensioning the arrays could lead to array bound errors in the earlier routines – the cost of an extra three real numbers per matrix is a small price to pay to avoid errors. Also note that when we DIMension an array it is immediately set to zero (see page 202 of the Spectrum BASIC Handbook (Vickers (1982)). We rewrite listings 4.1, 4.2, 4.3 and 4.4 as listings 4.1a, 4.2a, 4.3a and 4.4a respectively, to making use of these facts.

Listing 4.1a

```
9100 REM mult2
9101 REM IN  : A(3,3),R(3,3)
9102 REM OUT : R(3,3)
9110 FOR I = 1 TO 2
9120 FOR J = 1 TO 3
9130 LET B(I,J) = A(I,1)*R(1,J) + A(I,2)*R(2,J)
9140 NEXT J
```

```
9150 LET B(I,3) = B(I,3) + A(I,3)
9160 NEXT I
9170 FOR J = 1 TO 3
9180 LET R(1,J) = B(1,J): LET R(2,J) = B(2,J)
9190 NEXT J
9200 RETURN

9300 REM idR2
9302 REM OUT : R(3,3)
9310 DIM R(3,3)
9320 LET R(1,1) = 1: LET R(2,2) = 1
9330 RETURN
```

Listing 4.2a

```
9000 REM tran2
9001 REM IN  : TX,TY
9002 REM OUT : A(3,3)
9010 DIM A(3,3)
9020 LET A(1,1) = 1: LET A(2,2) = 1
9030 LET A(1,3) = TX: LET A(2,3) = TY
9040 RETURN
```

Listing 4.3a

```
8900 REM scale2
8901 REM IN  : SX,SY
8902 REM OUT : A(3,3)
8910 DIM A(3,3)
8920 LET A(1,1) = SX: LET A(2,2) = SY
8930 RETURN
```

Listing 4.4a

```
8600 REM rot2
8601 REM IN  : THETA
8602 REM OUT : A(3,3)
8610 DIM A(3,3)
8620 LET CT = COS THETA: LET ST = SIN THETA
8630 LET A(1,1) = CT: LET A(2,2) = CT
8640 LET A(1,2) = -ST: LET A(2,1) = ST
8650 RETURN
```

The construction of figure 4.2 may seem rather contrived since the position of the objects was chosen in an arbitrary way. However, in most diagrams the positioning of objects will be well defined, the values being implicit in the diagram required. See the example below.

Example 4.3
Write a program that draws an ellipse with major axis A, minor axis B and centred at the point (CX, CY). The major axis makes an angle θ with the positive x-direction. Note that the order of transformations is important: first rotate and then translate. If we wish to draw ellipses with major axis horizontal then we need not use matrices, we can stay with the routine set in exercise 2.5 using ideas

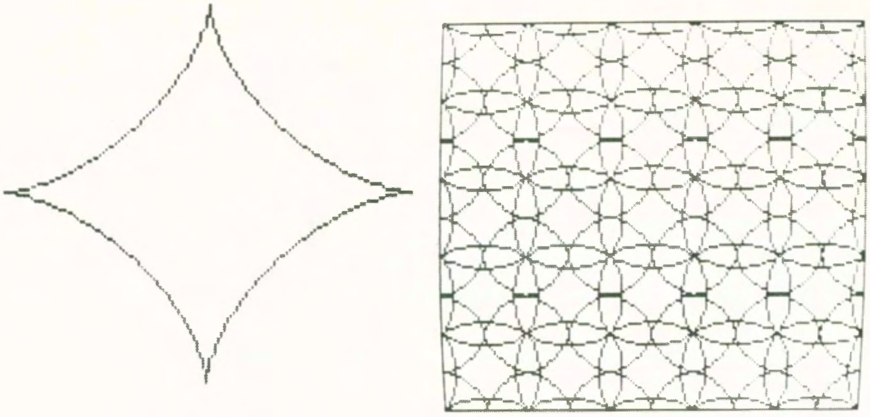
similar to those in listing 2.11a. Listing 4.10 gives a 'scene 2' routine that reads in data about the ellipse, calculates the SETUP to OBSERVED matrix and then calls the construction routine 'ellipse' that draws the ellipse.

Listing 4.10

```
6000 REM scene2/ellipse
6010 DIM A(3,3): DIM B(3,3): DIM R(3,3)
6020 LET ellipse = 6500
6030 INPUT "(CX,CY) ";CX;",";CY,,"A ";A;",B ";B;",THETA ";THETA
6040 LET THETA = -THETA
6049 REM ellipse centred at (CX,CY) and tilted through angle THETA.
6050 GO SUB idR2
6060 GO SUB rot2: GO SUB mult2
6070 LET TX = CX: LET TY = CY
6080 GO SUB tran2: GO SUB mult2
6090 GO SUB ellipse
6100 RETURN

6500 REM ellipse
6501 REM IN  : A,B,R(3,3)
6509 REM ellipse, major axis A, minor axis B.
6510 LET XPT = A*R(1,1) + R(1,3)
6520 LET YPT = A*R(2,1) + R(2,3)
6530 GO SUB moveto
6540 LET ALPHA = 0: LET ADIF = PI/100
6549 REM calculate points (XPT,YPT) on ellipse, in OBSERVED position.
6550 FOR I = 1 TO 200
6560 LET ALPHA = ALPHA + ADIF
6570 LET AA = A*COS ALPHA: LET BB = B*SIN ALPHA
6580 LET XPT = AA*R(1,1) + BB*R(1,2) + R(1,3)
6590 LET YPT = AA*R(2,1) + BB*R(2,2) + R(2,3)
6600 GO SUB lineto
6610 NEXT I
6620 RETURN
```

Exercise 4.4
Write a routine for drawing an individual matrix-transformable object (in this case an *astroid*, shown in figure 4.3a) and then use the matrix techniques to draw combinations of these objects (as in figure 4.3b). An astroid is a closed curve with parametric form ($R\cos^3\theta/\sin^3\theta$) where $0 \leqslant \theta \leqslant 2\pi$, R being the radius (the maximum distance from the centre of the object). The parameters needed by this routine are the radius of the astroid and the transforming matrix. Figure 4.3b is the combination of a large number of two different forms of the astroid. One has radius 1 and is not rotated, the other has radius $\sqrt{2}$ and is rotated through $\pi/4$ radians.

Exercise 4.5
Experiment with these matrix techniques. Write a subroutine that generates the matrix necessary to rotate points in space by an angle θ about an arbitrary point (X, Y) in space (not necessarily the origin). Also produce another routine that generates the matrix that will reflect points about the general line $ay = bx + c$. (Use the ideas given in example 4.2 for the production of ship (c).)


(a) *Figure 4.3* (b)

Storing Information about Scenes

We mentioned earlier that certain situations arise where we need to store all the information about a scene in a large data-base rather than lose the information on leaving the construction routine. Our data-base will consist of arrays X and Y, of length greater than or equal to NOV, the final number of vertices to be stored. (These vertices can be stored in the SETUP, ACTUAL or OBSERVED position: it depends on the context of the problem.) We also need to store information on lines, in a two-dimensional array L whose first index is 1 or 2, and whose second index is a number between 1 and a value greater than or equal to NOL, the final number of lines in the scene. The Ith line joins the two vertices with indices L(1,I) to L(2,I): hence this information is independent of position, it simply says which two vertices are joined by the Ith line. NOV and NOL are initialised in the 'scene2' routine and incremented in the construction routines.

We now no longer require construction routines to draw lines, we use them only to create the data-base of lines, vertices, etc. (transformed by the matrix R). After 'scene2' has constructed the final scene in memory it calls another routine 'drawit', which draws the final picture. The 'scene2' routine will be very similar to those mentioned earlier; for example, the routine for drawing figure 4.2 in this new way will be that given in listing 4.9 with the three minor changes listed below

```
6010 DIM X(20): DIM Y(20): DIM L(2,20)

6030 LET NOV=0 : LET NOL=0: LET ship= 6500 : LET drawit=7000

6330 GO SUB drawit: RETURN
```

This is used in conjunction with listing 4.11, which gives the 'ship' construction routine (which only sets up the data), and 'drawit' routine.

Listing 4.11

```
6500 REM ship/ stored
6501 REM IN  : NOV,NOL,R(3,3),X(NOV),Y(NOV)
6502 REM OUT : NOV,NOL,X(NOV),Y(NOV),L(2,NOL)
6510 DATA 3,0,0,0,-1,1,2,0,-1,-1,1,2,2,3,3,4,4,5,5,2
6520 RESTORE ship
6530 LET NV = NOV
6531 REM read vertices, and move into position using matrix R.
6540 FOR I = 1 TO 5
6550 READ XX,YY
6560 LET NOV = NOV + 1
6570 LET X(NOV) = XX*R(1,1) + YY*R(1,2) + R(1,3)
6580 LET Y(NOV) = XX*R(2,1) + YY*R(2,2) + R(2,3)
6590 NEXT I
6531 REM read and store line information
6600 FOR I = 1 TO 5
6610 READ L1,L2
6620 LET NOL = NOL + 1
6630 LET L(1,NOL) = L1 + NV: LET L(2,NOL) = L2 + NV
6640 NEXT I
6650 RETURN

7000 REM drawit
7001 REM IN  : NOL,X(NOV),Y(NOV),L(2,NOL)
7009 REM draw line by joining pairs of vertices.
7010 FOR I = 1 TO NOL
7020 LET L1 = L(1,I): LET L2 = L(2,I)
7030 LET XPT = X(L1): LET YPT = Y(L1): GO SUB moveto
7040 LET XPT = X(L2): LET YPT = Y(L2): GO SUB lineto
7050 NEXT I
7060 RETURN
```

Suppose we wish to produce different views of the same scene (again we use figure 4.2 as an example); that is, the same SETUP to ACTUAL matrices P, but different ACTUAL to OBSERVED matrices Q. The obvious solution is to create a data-base for the scene with the vertices in the ACTUAL position. Now for each new OBSERVED position we calculate Q and enter it into another 'drawit' routine (see listing 4.12 – different from listing 4.11), which transfers each vertex from its ACTUAL to OBSERVED position using Q, stores them in arrays V and W, and recalls them when they are required for drawing. When using this method to construct different views of figure 4.2, only the 'scene2' and 'drawit' routines differ from their earlier manifestations, and then only slightly. We give them in listing 4.12.

Listing 4.12

```
6000 REM scene2/ variable look2 ; 4 ships (stored)
6009 REM construct 4 ships stored in ACTUAL position.
6010 DIM X(20) : DIM Y(20): DIM V(20): DIM W(20): DIM L(2,20)
6020 DIM A(3,3): DIM B(3,3): DIM R(3,3)
6030 LET ship = 6500: LET drawit = 7000
6039 REM ship a).
6040 LET NOV = 0: LET NOL = 0: GO SUB idR2: GO SUB ship
6049 REM ship b).
6050 LET SX = 4: LET SY = 2
6060 GO SUB scale2: GO SUB mult2
6070 LET THETA = PI/6
6080 GO SUB rot2: GO SUB mult2
6090 LET TX = 6: LET TY = 4
6100 GO SUB tran2: GO SUB mult2
6110 GO SUB ship
6119 REM ship c).
6120 LET AY = -3: LET AY = 4
6130 GO SUB angle
6140 LET TX = 0: LET TY = 3
6150 GO SUB tran2: GO SUB mult2
6160 LET THETA = -THETA
6170 GO SUB rot2: GO SUB mult2
6180 LET SX = 1: LET SY = -1
6190 GO SUB scale2: GO SUB mult2
6200 LET THETA = -THETA
6210 GO SUB rot2: GO SUB mult2
6220 LET TX = 0: LET TY = -3
6230 GO SUB tran2: GO SUB mult2
6240 GO SUB ship
6249 REM ship d).
6250 GO SUB idR2
6260 LET TX = 6: LET TY = 4
6270 GO SUB tran2: GO SUB mult2
6280 LET THETA = PI/6
6290 GO SUB rot2: GO SUB mult2
6300 LET SX = 4: LET SY = 2
6310 GO SUB scale2: GO SUB mult2
6320 GO SUB ship
6329 REM loop through observation points.
6330 GO SUB idR2: GO SUB look2
6340 CLS: GO SUB drawit
6350 GO TO 6330
6360 RETURN

7000 REM drawit
7001 REM IN  : NOV,NOL,R(3,3),X(NOV),L(2,NOL)
7009 REM transform vertices from ACTUAL to OBSERVED position.
7010 FOR I = 1 TO NOV
7020 LET V(I) = X(I)*R(1,1) + Y(I)*R(1,2) + R(1,3)
7030 LET W(I) = X(I)*R(2,1) + Y(I)*R(2,2) + R(2,3)
7040 NEXT I
7049 REM draw lines by joining pairs of vertices.
7050 FOR I = 1 TO NOL
7060 LET L1 = L(1,I): LET L2 = L(2,I)
7070 LET XPT = V(L1): LET YPT = W(L1): GO SUB moveto
7080 LET XPT = V(L2): LET YPT = W(L2): GO SUB lineto
7090 NEXT I
7100 RETURN
```

Exercise 4.6
Construct a dynamic scene. With each new view the ships will move relative to one another in some well-defined manner. The observer also should move in some simple way; for example, the eye starts looking at the origin, five views later it is looking at the point (10, 10), and with each view the head is tilted a further 0.1 radian. You no longer need to INPUT the values of (DX, DY) and ALPHA into 'look3', instead they should be calculated by the program. After you have read chapter 13 you can place these five pictures in store and recall them as a 'movie' (if you have the 48K machine).

Exercise 4.7
Construct a scene that is a diagrammatic view of a room in your house – with schematic two-dimensional drawings of tables, chairs, etc. placed in the room. Each different type of object has its own construction routine, and the 'scene2' routine should read in data to place these objects around the room. Once the scene is set produce a variety of views, looking from various points and orientations.

Or you can set up a line-drawing picture of a map, and again view it from various orientations. The number of possible choices of scene is enormous!

Because we are using the 'clip' option in 'lineto' we can choose small values for HORIZ and VERT, which has the effect of the observer zooming up close to parts of a scene, and all external lines will be conveniently clipped off.

Complete Programs

We group the listings 3.4 ('angle'), 4.1a ('mult2' and 'idR2'), 4.2a ('tran2'), 4.3a ('scale'), 4.4a ('rot2'), 4.5 ('look2') and 4.6 ('main program') under the heading 'lib2'.

I. 'lib1', 'lib2', listings 4.7 ('scene2') and 4.8 ('ship'). Data required: HORIZ, VERT, DX, DY and ALPHA. Try 8, 5, 1, 1, 0.5. Keep any four of these values fixed and systematically make small changes in the other data value.
II. 'lib1', 'lib2', listings 4.9 ('scene2') and 4.8 ('ship'). Data required: HORIZ, VERT. Try 30, 20; 200, 200; 200, 150.
III. 'lib1', 'lib2' and listings 4.10 ('scene2' and 'ellipse'). Data required: HORIZ, VERT, CX, CY, A, B, THETA. Try 30, 20, 0, 0, 12, 9, 0. Again fix all but one of the values and change the remaining value systematically.
IV. 'lib1', 'lib2', listings 4.9 ('scene2' adjusted as per text) and 4.11 ('ship' and 'drawit'). Data required: as for II above.
V. 'lib1', 'lib2', listings 4.11 ('ship' but not 'drawit') and 4.12 ('scene2' and 'drawit'). Data required: HORIZ, VERT, DX, DY, ALPHA. Try 60, 40, 0, 0, 0. Systematically change each of the data values in turn.

5 *Character Graphics on the ZX Spectrum*

In the first chapter (listing 1.2) we saw how a small block or character may be generated by eight binary numbers. Such a block of 8 by 8 pixels is known as a *character block*. The Spectrum has three types of characters built in: the 96 character *standard* set, the 16 character *block graphics* set and the 21 character *user-defined graphics* set. The latter two sets are accessed from the keyboard in *graphics mode*. These characters are placed on the display when PRINT is used, so we must take a closer look at this operation.

The PRINT command allows us to place characters on any of the 22 lines of the upper screen, from top (0) to bottom (21), and at any of 32 positions along each line, from left (0) to right (31). These are the character blocks and each contains eight lines of eight pixels. Each line of eight pixels corresponds to a value stored in one memory location; each pixel corresponds to a binary digit in an eight-digit binary number. This can be represented by a decimal number between 0 and 255. In the store there is a table of characters. For any given character the PRINT command either finds the eight VALUEs from the table or calculates them, and copies them into the appropriate display-file memory locations. This has the effect of drawing the character on the screen.

The Standard Character Set

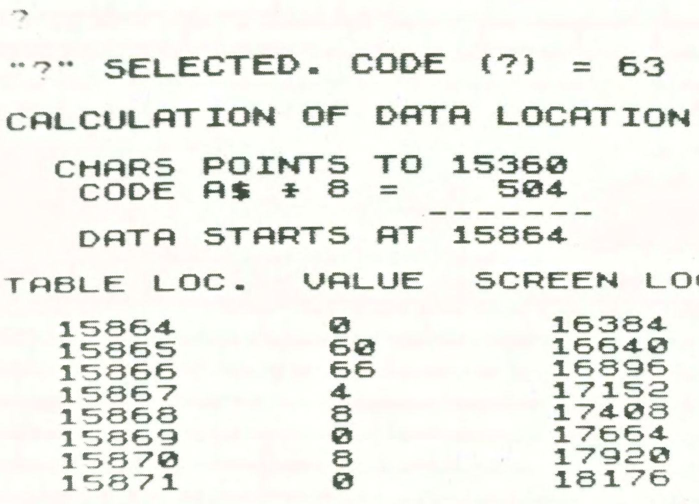
The table of data for the standard character set is stored in ROM, the permanent Read Only Memory of the computer. There are eight pieces of data for each of the 96 characters, thus the table consists of 768 (96*8) consecutive locations starting at 15616. Each character has a unique CODE number, see page 183 of the Spectrum BASIC Handbook (Vickers, 1982). The table contains the data for each of the characters in turn, starting with space (CODE 32) and ending with the copyright symbol (CODE 127). When the PRINT command requires the eight pieces of data for a given character, it looks at the system variable CHARS (see page 173 of the Spectrum BASIC Handbook (Vickers, 1982)), which contains the address of a location 256 (eight times the CODE for space) less than the start of the character table. In order to find the address of the first of the eight pieces of data it multiplies the CODE number of the character by eight and adds it to

CHARS. Finally PRINT copies the data to the display file and the character appears on the screen.

Run the following program, which is based on listing 1.2. It demonstrates how this process works by showing what calculations are performed. A detailed explanation of how to work out the display-file locations for any character block is given in chapter 13, but for the moment we shall continue to use block (0, 0) only. Figure 5.1 is an example run of this program using '?' as INPUT data.

Listing 5.1

```
 10 CLS: INPUT "Which character";A$: IF A$ = "" THEN GO TO 10
 20 LET A$ = A$(1): PRINT AT 2,0;"""";A$;"""" SELECTED. CODE (";A$;") = ";CODE A$
 30 PRINT AT 4,0;"CALCULATION OF DATA LOCATION"
 40 LET CHARS = PEEK 23606 + 256*PEEK 23607
 50 PRINT AT 6,2;"CHARS POINTS TO ";CHARS
 60 LET TABLE = CODE A$*8
 70 PRINT AT 7,3;"CODE A$ * 8 =     ";TABLE
 80 LET START = CHARS + TABLE
 90 PRINT AT 8,17;"-------": PRINT AT 9,2;" DATA STARTS AT ";START
100 LET CORNER = 16384: PRINT AT 11,0;"TABLE LOC.  VALUE  SCREEN LOC."
110 FOR I = 0 TO 7
120 LET VALUE = PEEK (START + I): LET MEMORY = CORNER + 256*I
130 PRINT AT I + 13,2;START + I: PRINT AT I + 13,22;MEMORY
140 PRINT AT I + 13,13;VALUE
150 POKE MEMORY,VALUE
160 NEXT I
170 INPUT "ANOTHER GO ? ";Y$: IF Y$ = "n" THEN STOP
180 GO TO 10
```



```
?
"?" SELECTED. CODE (?) = 63

CALCULATION OF DATA LOCATION

  CHARS POINTS TO 15360
   CODE A$ * 8 =     504
                 -------
   DATA STARTS AT 15864

TABLE LOC.  VALUE  SCREEN LOC.

  15864      0          16384
  15865      60         16640
  15866      66         16896
  15867      4          17152
  15868      8          17408
  15869      0          17664
  15870      8          17920
  15871      0          18176
```

Figure 5.1

Exercise 5.1

Rewrite listing 5.1 so that it will allow you to decide whether or not to accept each data VALUE before it is POKEd into the display file. If one of the eight

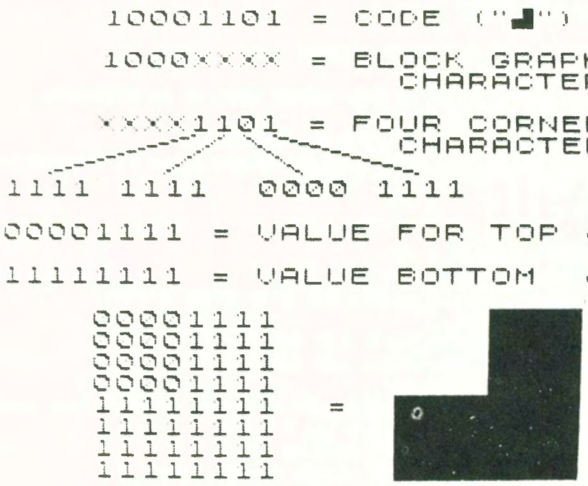
VALUEs is rejected, INPUT a replacement for this VALUE. Experiment by changing one or two VALUEs in a character.

The Graphics Mode

The address stored in CHARS is where you might expect to find data for the character with CODE 0; however, characters with CODEs between 0 and 31 are either control characters or unused. Since control characters are not displayed, then no data are stored for them. Although CHARS points to this address, the 256 (32*8) bytes following it are not used and so are available for other purposes. Furthermore, if a graphics mode character is selected for listing 5.1, then either blank or totally spurious data are gathered. This is because the standard table has data for CODEs 32 to 127 only and the graphics mode characters have CODEs greater than 127; so the program is looking in the wrong place, beyond the end of the table of data! The table in ROM immediately precedes the start of RAM, the Random Access Memory used for temporary storage, so the program will be looking at the display file that occupies the first 6K of RAM.

Block Graphics

The data for the block graphic set are not stored in ROM but instead are calculated from the CODE of the character. Each block graphics character has four quarters that are each represented, as on or off, by one of the last four binary digits of the CODE number. In a block graphics character the first four lines of eight pixels are identical, and the same is true of the bottom four lines.



```
        10001101  =  CODE ("▟")
        1000XXXX  =  BLOCK GRAPHICS
                        CHARACTER.
        XXXX1101  =  FOUR CORNERS OF
                        CHARACTER.
  1111 1111  0000 1111
  00001111  =  VALUE FOR TOP 4 LINES
  11111111  =  VALUE BOTTOM  4 LINES

        00001111
        00001111
        00001111
        00001111
        11111111     =
        11111111
        11111111
        11111111
```

Figure 5.2

Provided we know that a character is a block graphic, we can generate two pieces of data and use each four times to create the block. An example of how this is achieved is shown in figure 5.2.

The block graphics set can be used to give medium-resolution graphics of 64 by 44 quarter blocks. Using this idea we can build up quite complicated two-colour pictures very rapidly. The picture below (figure 5.3) was produced by the program given in listing 5.2. The program draws a small set of concentric circles in the top corner of the screen, then calls the 'big pixels' routine. This routine asks us to specify a character block that will be the top-left corner for our start position. From this position, for five character blocks down and for eight across, it stores the data from the display file in the array S. For each pair of lines down, it takes the eight pieces of data across and converts them to BINary strings (see line 1130 onwards). These strings are used to build up the BINary representation of the CODE for a graphics character by taking a two-bit section from each string in turn and prefacing it by the CODE for graphics character (line 1230).

Listing 5.2

```
 100 REM main program
 110 LET big pixels = 1000
 120 FOR I = 1 TO 18 STEP 3: CIRCLE 32,156,I: NEXT I
 130 GO SUB big pixels
 140 STOP

1000 REM big pixels
1010 DIM P(8): DIM S(40,8)
1020 FOR I = 1 TO 8: READ P(I): NEXT I: DATA 128,64,32,16,8,4,2,1
1030 DEF FN S(R,C) = 16384 + INT (R/8)*2048 + (R - INT (R/8)*8)*32 + C
1040 INPUT "TOP LEFT CORNER BLOCK IS AT     ROW,COL ";ROW;" , ";COL
1049 REM store display file data in array S for 5*8 blocks from ROW,COL.
1050 FOR I = 0 TO 4
1060 FOR J = 0 TO 7
1070 LET P = FN S(ROW + I,COL)
1080 FOR K = 0 TO 7
1090 LET S(I*8 + J + 1,K + 1) = PEEK(P + 256*J + K)
1100 NEXT K: NEXT J: NEXT I
1109 REM set screen to 20*32 which is four times area to be expanded.
1110 BORDER 1: PAPER 7: INK 0: CLS
1120 PRINT AT 21,0; PAPER 1;,,: PRINT AT 0,0; PAPER 1;,,
1129 REM take two lines of pixels at a time for translation to quarter blocks.
1130 FOR I = 1 TO 39 STEP 2
1140 LET A = I: LET B = I + 1
1150 FOR J = 1 TO 8
1160 LET A$ = "00000000": LET B$ = A$
1170 LET T = S(A,J): LET U = S(B,J)
1179 REM calculate binary forms of pixels for J'th block across.
1180 FOR K = 1 TO 8
1190 IF T >= P(K) THEN LET T = T - P(K): LET A$(K) = "1"
1200 IF U >= P(K) THEN LET U = U - P(K): LET B$(K) = "1"
1210 NEXT K
1219 REM take two pixels from each of the pair of bytes to make four quarters.
1220 FOR K = 1 TO 4: LET D = 2*K: LET C = D - 1
1230 LET C$ = "BIN 1000" + B$(C TO D) + A$(C TO D)
1239 REM convert binary form of quarter blocks to character.
1240 LET C = VAL C$: PRINT CHR$ C;
1250 NEXT K: NEXT J: NEXT I
1260 RETURN
```

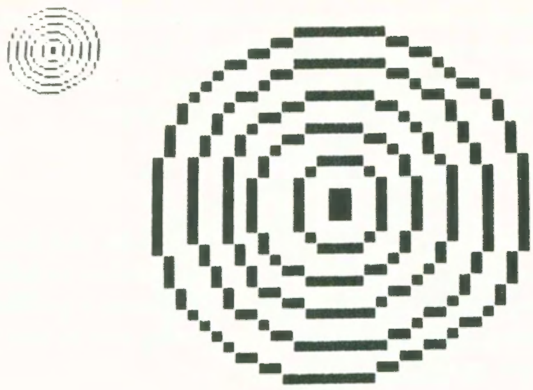


Figure 5.3

The graphics characters generated by the above program are printed on the screen and form an exact image of whatever was in the 5 by 8 rectangle of blocks specified. This image has been scaled in each direction by a factor of 4, so that it now covers 20 by 32 blocks and each original pixel is represented by a square of 4 by 4 pixels.

With the COPY facility and the ZX printer listings, four times the normal width and depth can be produced by using the 'big pixels' routine 16 times and then joining together the separate pieces of listing.

Exercise 5.2
Write a routine that produces a listing of itself by COPYing each 5 by 8 section as 'big pixels' to the printer.

User-defined Graphics

The third set of characters is the User-defined Graphics set (CODEs 144 to 164). Another table of data for this set is held in the last 168 locations at the end of memory. When the machine is turned on, the 21 characters from "A" to "U" are copied into this table. These characters are accessed from the keyboard in graphics mode by typing the letters "A" to "U". The data in the table for these characters can be changed by the user, so that any required character can be displayed directly from the keyboard. There is a built-in function USR, which when given a string argument ("A" to "U"), will give us the address of the first piece of data for the equivalent graphics character. The address of the start of this table is stored in the system variable UDG (see page 175 of the Spectrum BASIC Handbook (Vickers, 1982)). The CODE of the graphics character minus 144 and multiplied by 8 gives the relative position of the first of the eight pieces of data for that character in the table. The USR function calculates this position and adds it to UDG to get the absolute start address of the character in the store.

UDG can be altered so that fewer characters are available, say only those from A to H, leaving more room for programs, although this is usually unnecessary unless you have only 16K of RAM available. To illustrate the use of this function to alter a graphics character, type in the following command

POKE USR "A",BIN 00100100

Now if we enter graphics mode and type "A", we see that two extra pixel dots have appeared above the character "A". The program in listing 5.3 allows you to redefine all eight BINary VALUEs that represent a graphics character. The program then simulates graphics mode, allowing you to type user-defined graphics characters on the screen.

Listing 5.3

```
 10 INPUT "CHARACTER TO BE REDEFINED",U$: IF U$="" THEN GO TO 10
 20 IF U$ < "A" OR U$ > "U" THEN GO TO 10
 30 LET MEMORY = USR U$: CLS
 40 FOR I = 0 TO 7
 50 INPUT ("VALUE " + STR$ I +" = BIN ");LINE B$
 60 LET VALUE = VAL ("BIN " + B$)
 70 POKE MEMORY + I,VALUE
 80 IF LEN B$ < 8 THEN LET B$ = "0" + B$: GO TO 80
 90 PRINT B$: NEXT I
100 PRINT AT 10,0;"TYPE CHARACTERS OR ENTER"
110 REM simulate graphics mode for letters A to U.
110 LET R = 12: LET C = 0
120 PRINT AT R,C; FLASH 1;"G"
130 IF INKEY$ <> "" THEN GO TO 130
140 LET A$ = INKEY$: IF A$ = "" THEN GO TO 140
149 REM if 'enter' is pressed restart program.
150 IF A$ = CHR$ 13 THEN GO TO 10
160 IF A$ < "a" OR A$ > "u" THEN GO TO 130
169 REM convert character to equivalent user-defined graphic and print.
170 LET A$ = CHR$ (CODE A$ + 47)
180 PRINT AT R,C;A$
189 REM move fake cursor pointers.
190 LET C = C + 1: IF C = 32 THEN LET C = 0: LET R = R + 1
    : IF R = 22 THEN LET R = 12
200 PRINT AT R,C; FLASH 1;"G"
210 GO TO 130
```

Exercise 5.3

Use listing 5.3 to generate a graphics character consisting of a chequer-board pattern of pixels instead of completely filled areas. Experiment with this using various PAPER and INK colours in order to make new colours, for example, red and yellow give orange.

User-defined graphics characters can be directly incorporated in your programs to enhance the speed of display construction. The equivalent character could be constructed by a series of DRAW and PLOT commands but this would take longer in most cases. For instance, with characters we can build up patterns on the screen very quickly compared to the length of time it would take to PLOT or DRAW the same pattern. Listing 5.4 shows how this can be done using user-defined characters "A" to "D" (the darker characters on the listing denote characters in graphics mode). When these graphics characters have been defined then they, rather than the corresponding alphabetic graphics character, will appear in program listings.

Listing 5.4

```
200 REM main program
209 REM fill screen with a diagonal pattern of first four graphics characters.
210 OVER 0
220 INPUT " INK ";I: INK I
230 INPUT " PAPER ";I: PAPER I
240 LET A$ = "ABCD": LET B$ = "BCDA"
250 LET C$ = "CDAB": LET D$ = "DABC"
260 FOR J = 0 TO 7
270 FOR I = 0 TO 4
280 PRINT AT I*4,J*4;A$
290 PRINT AT I*4 + 1,J*4;B$
300 PRINT AT I*4 + 2,J*4;C$
310 PRINT AT I*4 + 3,J*4;D$
320 NEXT I
330 PRINT AT I*4,J*4;A$
340 PRINT AT I*4 + 1,J*4;B$
350 NEXT J
360 STOP
```

Alternate Character Sets

So far we have used up to 37 graphics characters to create pictures, but by using alternate standard sets we can add a further 96 with every new set we define. To use an alternate set we simply change the value stored in CHARS so that it points to our new set in RAM instead of the normal table. We have seen that the amount of work that goes into constructing even one character is enough to imply that the construction of a complete new character set will be a very arduous task. So we need a program that will simplify this task, and also allow characters to be altered once created. Listing 5.5 is such a character-generation and editing program, and was designed for use in the development of graphical display programs.

Listing 5.5

```
     1 CLEAR 62294: INK 0: PAPER 7: BORDER 7: OVER 1: LET N = 0
: REM for 16K machines use CLEAR 255 + the value of S(I), where I is the index
     of thelowest character set which  will fit in the available store.
     2 DIM Z$(8,8): DIM T$(8,8): DIM S(6): DIM P(8): DIM B(8): DIM G(7)
: REM line numbers to GO TO for menu options.
     3 FOR I = 1 TO 7: READ G(I): NEXT I
       : DATA 61,67,71,100,106,112,119
: REM powers of 2 ready for use in binary conversions.
     4 FOR I = 1 TO 8: READ P(I): NEXT I
       : DATA 128,64,32,16,8,4,2,1
: REM values which must be placed in CHARS to use each alternative set.
  for 16K machines these numbers should be,
  15360,29271,30039,30807,31575,32080 (see Appendix).
      5 FOR I = 1 TO 6: READ S(I): NEXT I
        : DATA 15360,62039,62807,63575,64343,64848
      6 LET S =1 : GO SUB 17
: REM check set 1 is being used and print menu.
      7 CLS : PRINT AT 2,2;"*** CHARACTER GENERATOR ***"
      8 PRINT AT 5,6; PAPER 5;"1"; PAPER 7;" ... PRINT ALL SETS"
      9 PRINT AT 7,6; PAPER 5;"2"; PAPER 7;" ... PRINT ONE SET"
     10 PRINT AT 9,6; PAPER 5;"3"; PAPER 7;" ... EDIT CHARACTER"
     11 PRINT AT 11,6; PAPER 5;"4"; PAPER 7;" ... COPY SET TO SET"
     12 PRINT AT 13,6; PAPER 5;"5"; PAPER 7;" ... SAVE SET"
     13 PRINT AT 15,6; PAPER 5;"6"; PAPER 7;" ... LOAD SET"
     14 PRINT AT 17,6; PAPER 5;"7"; PAPER 7;" ... RUN YOUR PROGRAM"
: REM wait for valid option.
     15 INPUT "WHICH OPTION ";OP: IF OP < 1 OR OP > 7 THEN GO TO 15
: REM jump to appropriate routine.
     16 GO TO G(OP)
: REM most subroutines are located at start of program for efficiency.
: REM routine to change to set S by altering chars to point to new table of data.
     17 LET HI = INT (S(S)/256): LET LO = S(S) - HI*256
     18 POKE 23606,LO: POKE 23607,HI: RETURN
: REM general routine to wait before clearing screen and returning to menu.
     19 LET S = 1: GO SUB 17: INPUT "PRESS ENTER TO RETURN TO MENU"; LINE A$ : RE
: REM clear binary characters array.
     20 FOR I = 1 TO 8: LET Z$(I) = "00000000": NEXT I: RETURN
: REM input a valid set number and give upper and lower bounds for characters.
    21 INPUT "WHICH SET ";S: IF S < 1 OR S > 6 THEN GO TO 21
     22 LET A = 32: LET B = 127: IF S = 6 THEN LET A = 65: LET B = 85
     23 RETURN
: REM input a valid character for chosen set.
     24 INPUT "WHICH CHARACTER ";C$: LET C = CODE C$
        : IF C < A OR C > B THEN GO TO 24
     25 RETURN
: REM draw eight by eight block grid at horizontal position altered by N.
     26 LET NN = 64 + N: FOR I = 0 TO 8
     27 PLOT I*8 + NN,64: DRAW 0,64
     28 PLOT NN,I*8 + 64: DRAW 64,0
     29 NEXT I: RETURN
: REM produce graphical equivalent of binary string array inside grid.
     30 FOR I = 1 TO 8: FOR J = 1 TO 8
     31 LET P = 7: IF N = 0 AND Z$(I,J) = "1" OR N = 96 AND T$(I,J) = "1"
        THEN LET P = 4
     32 GO SUB 48: NEXT J: NEXT I: RETURN
: REM overprint cursor on a square in the grid.
     33 PLOT X,Y-2: DRAW 0,4
     34 PLOT X-2,Y: DRAW 4,0
     35 RETURN
: REM double a corner of a character into a temporary array T$ and save it.
     36 INPUT "WHICH CORNER ";C: IF C < 1 OR C > 4 THEN GO TO 36
     37 FOR I = 1 TO 8: LET T$(I) = "00000000": NEXT I: GO TO 36 + C*2
     38 FOR I = 1 TO 4: FOR J = 1 TO 4: IF Z$(I,J) = "1"
        THEN LET TI = 2*I: LET TJ = 2*J: GO SUB 46
     39 NEXT J: NEXT I: RETURN
     40 FOR I = 1 TO 4: FOR J = 5 TO 8: IF Z$(I,J) = "1"
        THEN LET TI = 2*I: LET TJ = 2*(J - 4) : GO SUB 46
     41 NEXT J: NEXT I: RETURN
     42 FOR I = 5 TO 8: FOR J = 1 TO 4: IF Z$(I,J) = "1"
        THEN LET TI = 2*(I - 4): LET TJ = 2*J: GO SUB 46
     43 NEXT J: NEXT I: RETURN
    44 FOR I = 5 TO 8: FOR J = 5 TO 8: IF Z$(I,J) = "1"
       THEN LET TI = 2*(I - 4): LET TJ = 2*(J - 4): GO SUB 46
    45 NEXT J: NEXT I: RETURN
: REM subroutine to convert one pixel into four pixels in doubled array.
    46 LET T$(TI,TJ) = "1": LET T$(TI-1,TJ) = "1": LET T$(TI,TJ-1) = "1"
       : LET T$(TI-1,TJ-1) = "1": RETURN
: REM overprint the identifying labels at the corners of the grid.
    47 PRINT AT 5,7;"1": PRINT AT 5,16;"2": PRINT AT 14,7;"3"
       : PRINT AT 14,16;"4": RETURN
: REM subroutine to print a coloured block in a square of the grid.
    48 PRINT AT I+5,J+7+N/8; PAPER P;" ": RETURN
: REM display doubled character in a grid on the right and save character.
    49 LET N = 96: GO SUB 26: GO SUB 30
    50 GO SUB 21: GO SUB 24: LET D = S(S) + C*8 - 1
    51 FOR I = 1 TO 8: POKE D + I,VAL ("BIN " + T$(I)): NEXT I
    52 LET N = 0: CLS: RETURN
: REM copy reflection of Z$ array about x-axis into T$ array.
    53 FOR I = 1 TO 8: FOR J = 1 TO 8
    54 LET T$(9-I,J) = Z$(I,J): NEXT J: NEXT I: GO SUB 59: RETURN
: REM copy rotation of Z$ array into T$ array.
    55 FOR I = 1 TO 8: FOR J = 1 TO 8
    56 LET T$(9-J,I) = Z$(I,J): NEXT J: NEXT I: GO SUB 59: RETURN
: REM copy reflection of Z$ array about y-axis into T$ array.
    57 FOR I = 1 TO 8: FOR J = 1 TO 8
    58 LET T$(I,9-J) = Z$(I,J): NEXT J: NEXT I: GO SUB 59: RETURN
: REM routine to copy T$ array back into Z$ array.
    59 FOR I = 1 TO 8: FOR J = 1 TO 8
    60 LET Z$(I,J) = T$(I,J): NEXT J: NEXT I: RETURN
: REM start of option handling routines.
: REM print out all five complete character sets and user-defined graphics set.
    61 CLS: FOR S = 1 TO 5: GO SUB 17
    62 PRINT AT S*4 - 4,0;
    63 FOR C = 32 TO 127: PRINT CHR$ C;: NEXT C
    64 NEXT S: GO SUB 17: PRINT AT 20, 0;
    65 FOR C = 65 TO 85: PRINT CHR$ C;: NEXT C
: REM use standard routine to wait then back to menu.
    66 GO SUB 19: GO TO 7
: REM print out one set and give the charcters with which they correspond.
    67 CLS: GO SUB 21: LET SS = S
    68 FOR C = A TO B: PRINT PAPER 6;CHR$ C;"=";: LET S = SS: GO SUB 17
    69 PRINT CHR$ C;: LET S = 1: GO SUB 17: PRINT " ";: NEXT C
: REM return to menu.
    70 GO SUB 19: GO TO 7
: REM option three, editting a character, clear screen and Z$ binary array.
    71 CLS: GO SUB 20
: REM which set and character, calculate data position for character.
    72 GO SUB 21: GO SUB 24: LET D = S(S) + C*8 - 1
: REM get eight binary numbers from table of data.
    73 FOR I = 1 TO 8: LET B(I) = PEEK (D + I): NEXT I
: REM convert each number into a binary string of eight digits.
    74 FOR I = 1 TO 8: LET T = B(I)
    75 FOR J = 1 TO 8
    76 IF T >= P(J) THEN LET Z$(I,J) = "1": LET T = T - P(J)
    77 NEXT J: NEXT I
: REM draw a grid and put green blocks in squares to represent binary ones.
    78 GO SUB 26: GO SUB 30
: REM initialise cursor position in pixels and grid reference values.
    79 LET X = 68: LET Y = 124: LET I = 1: LET J = 1
: REM overprint cursor on square.
    80 GO SUB 33
```

```
: REM wait for next keypress.
   81 IF INKEY$ <> "" THEN GO TO 81
   82 IF INKEY$ = "" THEN GO TO 82
: REM get command and remove cursor.
   83 LET A$ = INKEY$: BEEP 0.05,30: GO SUB 33
: REM check for cursor movement controls with or without the shift key.
   84 IF (A$ = "5" OR A$ = CHR$ 8) AND J > 1 THEN LET X = X - 8: LET J = J - 1
   85 IF (A$ = "6" OR A$ = CHR$ 10) AND I < 8 THEN LET Y = Y - 8: LET I = I + 1
   86 IF (A$ = "7" OR A$ = CHR$ 11) AND I > 1 THEN LET Y = Y + 8: LET I = I - 1
   87 IF (A$ = "8" OR A$ = CHR$ 9) AND J < 8 THEN LET X = X + 8: LET J = J + 1
: REM check for special commands, in upper or lower case.
: REM Plot, fill in a square.
   88 IF A$ = "P" OR A$ = "p" THEN LET  P = 4: LET Z$(I,J) = "1": GO SUB 48
: REM Off, blank out a square.
   89 IF A$ = "O" OR A$ = "o" THEN LET  P = 7: LET Z$(I,J) = "0": GO SUB 48
: REM Rotate character by 90 degrees anti-clockwise.
   90 IF A$ = "R" OR A$ = "r" THEN GO SUB 55: GO SUB 30: GO TO 79
: REM reflect in the X or horizontal axis.
   91 IF A$ = "X" OR A$ = "x" THEN GO SUB 53: GO SUB 30: GO TO 79
: REM reflect in the Y or vertical axis.
   92 IF A$ = "Y" OR A$ = "y" THEN GO SUB 57: GO SUB 30: GO TO 79
: REM Merge another character with current pattern.
   93 IF A$ = "M" OR A$ = "m" THEN GO SUB 26: GO TO 72
: REM make Double-size copy of one corner of character.
   94 IF A$ = "D" OR A$ = "d" THEN GO SUB 47: GO SUB 36: GO SUB 49: GO TO 78
: REM unless you want to Save the character go back and wait for more commands.
   95 IF A$ <> "S" AND A$ <> "s" THEN GO TO 80
: REM replace character in data table by using the BIN function to convert.
   96 GO SUB 21: GO SUB 24
   97 LET D = S(S) + C*8 - 1
   98 FOR I = 1 TO 8: POKE D + I,VAL("BIN " + Z$(I)): NEXT I
: REM return to menu.
   99 GO SUB 19: GO TO 7
: REM option four copy set to set.
  100 INPUT "FROM SET ";A;" TO SET ";B
: REM make sure sets are valid.
  101 IF A < 1 OR B < 1 OR A > 5 OR B > 5 THEN GO TO 100
: REM don't bother if sets are same.
  102 IF A = B THEN GO TO 105
: REM make copy of data from one table to another.
  103 PRINT AT 21,10;"PLEASE WAIT": FOR K = 256 TO 1024
  104 POKE (S(B) + K), PEEK (S(A) + K): NEXT K
: REM return to menu.
  105 GO SUB 19: GO TO 7
: REM option five save data table for a set of characters as bytes on tape.
  106 CLS: PRINT AT 2,6;"SAVING CHARACTERS": INPUT "WHAT FILE NAME ";N$
      : IF N$ = "" THEN GO TO 7
  107 PRINT AT 4,10;N$: INPUT "SAVE WHICH SET ";S
      : IF (S < 2 OR S > 6) AND S <> 11 THEN GO TO 107
: REM if you type 5+6 then spectrum sets S = 11 so save sets 5 and 6 together.
  108 IF S = 11 THEN SAVE N$ CODE (S(5) + 256),168 + 768: GO TO 111
: REM set 6 is smaller than normal sets.
  109 IF S = 6 THEN SAVE N$ CODE (S(6) + 512),168: GO TO 111
  110 SAVE N$ CODE (S(S) + 256),768
: REM return to menu.
  111 GO SUB 19: GO TO 7
: REM option six reload data table for a set of characters as bytes from tape.
  112 CLS: PRINT AT 2,6;"LOADING CHARACTERS": INPUT "WHAT FILE NAME ";N$
      : IF N$ = "" THEN GO TO 7
  113 PRINT AT 4,10;N$: INPUT "LOAD WHICH SET ";S
      : IF (S < 2 OR S > 6) AND S <> 11 THEN GO TO 113
  114 PRINT AT 21,10;"Start tape."
```

```
: REM if you type 5+6 then spectrum sets S = 11 so load sets 5 and 6 together.
   115 IF S = 11 THEN LOAD N$ CODE (S(5) + 256),168 + 768: GO TO 118
: REM load user defined set.
   116 IF S = 6 THEN LOAD N$ CODE (S(6) + 512),168: GO TO 118
   117 LOAD N$ CODE (S(S) + 256),768
: REM return to menu.
   118 GO SUB 19: GO TO 7
```

The CHARACTER GENERATOR routines are intended to take all the hard work out of preparing and using defined characters: they allow you to edit and manipulate characters, use alternate character sets and graphics characters, save and reload defined chracters and immediately test your own programs. The routines are designed for the 48K Spectrum to sit in the bottom part of the memory (lines 1 to 118), followed by your own program starting after line 118, followed by the renumber and delete programs (from chapter 13) starting at line 9900. If you have the 16K version, then this program should be run independently of other routines – see Appendix A.

The program first offers a choice of seven options, which we will look at in turn.

(1) The first option is PRINT ALL SETS. This will print the normal character set (set 1), followed by the four alternate character sets (2 to 5), followed by the user-defined graphics set (set 6). The last few characters of set 5 will contain spurious data, a remnant of the GO SUB stack, which has been moved out of harm's way.
(2) The second option is PRINT ONE SET. These are identified with the numbers 1 to 6 as follows: 1 standard set – cannot be changed; 2 to 5 alternate sets – may temporarily replace standard set; 6 user-defined graphics set – always available. On selecting one of these numbers the screen will display each character of the standard set with = after it, both on a yellow background, followed by the equivalent character of the alternate set. For set 6 this will be produced only for the characters "A" to "U", and it will show the characters available when using these keys in graphics mode.
(3) Option three is the editor. This is the most complicated option and has a large set of commands accessed by typing their initial letter. First, though, you will be asked with which set and which character you wish to start. If you want to start with a blank grid of eight by eight pixels, use set 1 and the space character. (To use the quote marks character it will be necessary to type it twice – see page 47 of the Spectrum BASIC Handbook (Vickers, 1982)).

You will now be in edit mode and the character you have chosen will be displayed as green blocks in an 8 by 8 grid. The cross in the top-left corner is the edit cursor. The cursor is controlled by the standard cursor keys ("5", "6", "7", "8") either with or without the shift key.

The first two commands are PLOT or OFF, which change a square in the character grid. Simply move the cursor over the square and press either "P" or "O".

"P" makes the square green, equivalent to a binary on or one INKed-in pixel, and "O" erases the square to white.

The next three commands all specify transformations similar to those performed on two-dimensional objects in chapter 4. ROTATE turns the character through 90 degrees anti-clockwise about its centre. X-AXIS reflects the character about the horizontal axis and Y-AXIS reflects the character about the vertical. These commands can be used to create text sets for any orientation.

Finally we have MERGE, SAVE and DOUBLE. MERGE allows any character to be merged, into the grid, on top of what is already being edited. This is very useful for creating foreign language sets; for example, by placing a slash through an O for Scandinavian languages, or by adding accents for French, etc. to letters. The SAVE command asks in which set we wish to save the newly created character, and to which standard character is it equivalent. If set 1 is specified the character will be lost, since set 1 is held in the Read Only Memory of the Spectrum: this can be useful to dispose of any unwanted grids created by mistake. DOUBLE is a powerful feature that allows you to take any quarter of the character you are editing, create a double-size copy of the quarter and SAVE it immediately as a single character. This option does not affect the character you are editing, so all four corners may be copied in succession into four different characters within the same editing stage. This feature can be used to create characters of double, quadruple or larger size with ease.

(4) COPY SET TO SET is the fourth option available and can be used to make copies of a complete set for subsequent manipulation, or simply to move one of the alternate sets around in memory. Sets 2 to 5 are stored at the end of memory, in the locations preceding the user-defined graphics, with set 5 ending at the location immediately before the user-defined set. The area of store used for sets 2 to 5 is reserved and protected from BASIC use by the command CLEAR 62294 in the 48K version (see page 168 of the Spectrum BASIC Handbook (Vickers, 1982)). 62294 is one less than the first location in the table for set 2. In order to protect only the areas for sets I onwards, we need to evaluate S(I) + 255 and place this number in the CLEAR command at the start of the program. S(I) is the value assigned to the system variable CHARS that enables PRINT to use set I. These values are READ from the DATA statement at line 3. Note the changes for the 16K machine given in Appendix A.

(5) and (6) Option five allows character sets to be saved on tape and option six for them to be reloaded. If, in reply to the question 'WHICH SET', we type a number between 1 and 6, then that set is SAVEd or reLOADed. Should we type 5 + 6 then both sets 5 and 6 will be saved or loaded as a single unit.

To allow other programs to load and use alternate sets of characters, lines 112 to 117 should be copied from the 'CHARACTER GENERATOR' along with the subroutine (at 17 and 18), which allows switching between sets. The array S and its DATA from line 5 must also be included.

(7) The final option is RUN YOUR PROGRAM, which will transfer control to the first line following the character generator or, if nothing is there, stop.

In order to familiarise yourself with the 'CHARACTER GENERATOR' follow the instructions laid out below.

MERGE listings 5.4 and 5.5 on the 48K Spectrum; run them separately on the 16K machine. Create a character like an ink-blot pattern and SAVE it as character "**A**" in set 6. Edit the character: rotate it and SAVE it as character "**B**" in set 6. Edit "**A**" in set 6: use X-AXIS reflection on it and SAVE it as "**C**" in set 6. Edit "**B**" in set 6: use Y-AXIS reflection on it and SAVE it as "**D**" in set 6. Now use option 7 to RUN YOUR PROGRAM and see what pattern emerges.

Exercise 5.4
Experiment with various possible symmetries of characters and patterns for placement of characters on the screen. Alter the program so that it tries all the possible combinations of INK and PAPER colours within two nested FOR. . . NEXT loops.

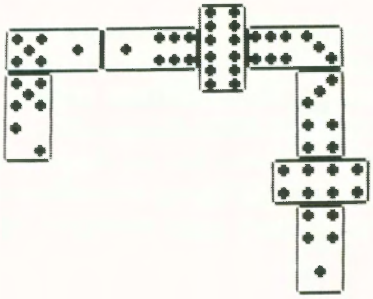
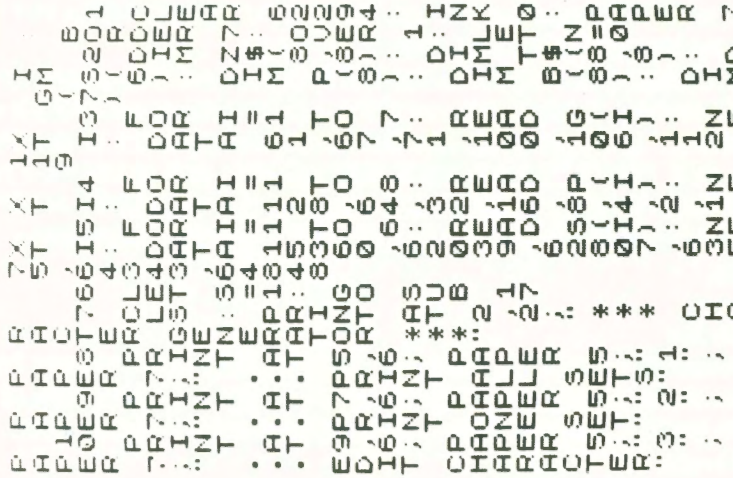


Figure 5.4

Exercise 5.5
Now create characters for dominoes using the 'CHARACTER GENERATOR' from listing 5.5. This enables construction of a display similar to figure 5.4.

Exercise 5.6
Write a routine that copies one set to another but uses some of the editor routines from the 'CHARACTER GENERATOR' to perform transformations on each character as it is copied. Figure 5.5 shows a listing produced with a set that has been ROTATED.

Let us now consider a complete program that has been developed using the 'CHARACTER GENERATOR' and subsequently separated from the development system to stand alone. The 'MASTER MIND' program (listing 5.6) is shown in mid-game in figure 5.6 and we can see immediately that an alternate character



```
   1 CLEAR 62294: INK 0: PAPER 7:
 BORDER 7: OVER 1: LET N=0
   2 DIM N$(8,8): DIM T$(8,8): D
IM S(6): DIM P(8): DIM B(8): DIM
 G(7)
   3 FOR I=1 TO 7: READ G(I): NE
XT I: DATA 61,67,71,100,106,112,
119
   4 FOR I=1 TO 8: READ P(I): NE
XT I: DATA 128,64,32,16,8,4,2,1
   5 FOR I=1 TO 6: READ S(I): NE
XT I: DATA 15360,62039,62807,635
75,64343,64848
   6 LET S=1: GO SUB 17
   7 CLS : PRINT AT 2,2;"*** CHA
RACTER GENERATOR ***"
   8 PRINT AT 5,6; PAPER 5;" 1: ";
PAPER 7;" ... PRINT ALL SETS"
   9 PRINT AT 7,6; PAPER 5;" 2: ";
PAPER 7;" ... PRINT ONE SET"
  10 PRINT AT 9,6; PAPER 5;" 3: ";
PAPER 7;" ... EDIT CHARACTER"
```

Figure 5.5

set was used. This character set was created by DOUBL(E)ing the required letters and numerals of set 1 and storing the top-left and top-right corners in the capital letters of set 5 and the bottom corners in the lower case letters of set 5. The two sizes of peg were stored in the user-defined graphics set, which is independent of the main set; hence they are always available whether we are using set 1 or set 5. The DOUBLE letters were edited to smooth out their edges and the combined sets 5 + 6 were stored on cassette tape as 'masterset'.

Listing 5.6

```
      1 CLEAR 62294: INK 0: PAPER 7: BORDER 7
  : REM routines to allow use of alternate sets
  : REM remember to make alterations for 16K machines as in listing 5.5.
      2 DIM S(6)
      5 FOR I = 1 TO 6: READ S(I): NEXT I
        : DATA 15360,62039,62807,63575,64343,64848
      6 LET S = 1 : GO SUB 17
      7 GO TO 200
     17 LET HI = INT (S(S)/256): LET LO = S(S) - HI*256
     18 POKE 23606,LO: POKE 23607,HI: RETURN
    112 CLS: PRINT AT 2,6;"LOADING CHARACTERS": LET N$ = "masterset"
    113 PRINT AT 4,10;N$: LET S = 5 + 6
    114 PRINT AT 21,10;"Start tape.": PRINT AT 5,0;
    115 LOAD N$ CODE (S(5) + 256),168 + 768
    118 RETURN

    198 REM load characters and initialise scores
    199 REM arrays to hold Guess, Target and a copy of target for checking.
    200 GO SUB 112: LET MYSCORE = 0: LET SCORE = 0: DIM G(5): DIM T(4): DIM X(4)
    209 REM use routine to draw display.
    210 RANDOMIZE: GO SUB 480
    219 REM set up target colours and go into guess loop.
    220 FOR I = 1 TO 4: LET T(I) = INT (RND*6) + 1: NEXT I: LET GO = 0
    229 REM next guess, if you have had six then you lose.
```

```
230 LET GO = GO + 1: IF GO = 7 THEN GO TO 390
239 REM input routine for guess.
240 GO SUB 780
249 REM print guess on board (A is graphics A).
250 PAPER 7
260 PRINT AT 1 + GO*3,11; INK G(1);"A"; INK G(2);"  A";
    INK G(3);"  A"; INK G(4);"  A"
269 REM check guess against temporary copy of target.
270 LET NB = 0: LET NW = 0: FOR I = 1 TO 4: LET X(I) = T(I): NEXT I
279 REM look for exact matches first, if found then cross off both pegs.
280 FOR I = 1 TO 4: IF G(I) = X(I) THEN LET X(I) = 0: LET NB = NB + 1
    : LET G(I) = 7
290 NEXT I
299 REM check for right colour wrong position.
300 FOR I = 1 TO 4: FOR J = 1 TO 4: IF G(I) = X(J) THEN LET NW = NW + 1
    : LET X(J) = 0: LET G(I) = 7
310 NEXT J: NEXT I
319 REM set up array x with appropriate numbers of black and white pegs.
320 FOR I = 1 TO NB: LET X(I) = 0: NEXT I: FOR I = NB + 1 TO NB + NW
329 REM pad out array with non-visible green pegs.
330 LET X(I) = 7: NEXT I: IF NB + NW < 4 THEN FOR I = NB + NW + 1 TO 4
    : LET X(I) = 4: NEXT I
339 REM display marker pegs on board.
340 PAPER 4: PRINT AT GO*3 + 1,2; INK X(1);"B"; INK X(2);"B";
350 PRINT INK X(3);"B"; INK X(4);"B";
359 REM if you got it all right then you win.
360 IF NB = 4 THEN GO TO 400
369 REM add one to computers score and loop back for next guess.
370 LET MYSCORE = MYSCORE + 1: PRINT AT 10,26; PAPER 6; INK 0;MYSCORE
380 GO TO 230
389 REM if you lose computer gets 10 more points.
390 LET MYSCORE = MYSCORE + 10: GO TO 410
399 REM if you win you get 10 points.
400 LET SCORE = SCORE + 10
409 REM build up string for display at bottom.
410 LET I$ = CHR$ 17 + CHR$ 7 + "TARGET WAS " + CHR$ 16 + CHR$ T(1) + "A  "
420 LET I$ = I$ + CHR$ 16 + CHR$ T(2) + "A  "
430 LET I$ = I$ + CHR$ 16 + CHR$ T(3) + "A  "
440 LET I$ = I$ + CHR$ 16 + CHR$ T(4) + "A" + CHR$ 16 + CHR$ 0
450 LET I$ = I$ + CHR$ 6 + " PRESS ENTER TO CONTINUE        "
459 REM show target and wait for 'enter' before restarting game.
460 INPUT (I$); LINE B$
470 GO TO 210
479 REM draw board.
480 OVER 0: BORDER 6: PAPER 7: CLS: INK 0
489 REM use alternate set to print MASTER MIND in double size.
490 LET S = 5: GO SUB 17
500 PRINT AT 0,0;" ABCDEFGHIJKL  ABMNOPQR"
510 PRINT AT 1,0;" abcdefghijkl  abmnopqr"
519 REM print out double size numbers.
520 PRINT AT 4,7;"ST": PRINT AT 5,7;"st"
530 PRINT AT 7,7;"UV": PRINT AT 8,7;"uv"
540 PRINT AT 10,7;"WX": PRINT AT 11,7;"wx"
550 PRINT AT 13,7;"YZ": PRINT AT 14,7;"yz"
560 PRINT AT 16,7;"[\": PRINT AT 17,7;"{|"
570 PRINT AT 19,7;"]↑": PRINT AT 20,7;"}~"
579 REM wipe of sides of board in yellow.
580 PAPER 4: FOR I = 3 TO 20: PRINT AT I,1;"      ": NEXT I
590 PAPER 6: FOR I = 0 TO 21: PRINT AT I,24;"        ": NEXT I
600 PAPER 7: FOR I = 4 TO 19 STEP 3
609 REM print holes for pegs.
610 PRINT AT I,11;"A  A  A  A": NEXT I
```

```
619 REM print credits on side.
62Ø LET S = 1: GO SUB 17: PAPER 2
63Ø LET G$ = "      c BY  BRIAN JONES  AND   IAN  ANGELL 1982        "
64Ø FOR I = Ø TO 8
65Ø PRINT AT 12 + I,24; PAPER 6; INK 2; CHR$ 133;
    PAPER 2; INK 7;G$(I*6 + 1 TO I*6 + 6);
66Ø PRINT INK 2; PAPER 6;CHR$ 138: NEXT I
669 REM set up score displays.
67Ø PAPER 6: PRINT AT 2,26;"YOUR": PRINT AT 3,25;"SCORE"
68Ø PRINT AT 5,26;SCORE
69Ø PRINT AT 7,26;"MY": PRINT AT 8,25;"SCORE"
7ØØ PRINT AT 1Ø,26;MYSCORE
7Ø9 REM draw horizontal lines across board.
71Ø FOR I = 2 TO 2Ø STEP 3: LET Y = 168 - 8*I
72Ø PLOT 7,Y: DRAW 176,Ø: PLOT 7,Y - 1: DRAW 176,Ø: NEXT I
729 REM draw vertical lines on board.
73Ø PLOT 7,7: DRAW Ø,145: PLOT 8,7: DRAW Ø,145
74Ø PLOT 183,7: DRAW Ø,145: PLOT 184,7: DRAW Ø,145
75Ø PLOT 55,7: DRAW Ø,145: PLOT 56,7: DRAW Ø,145
76Ø PLOT 71,7: DRAW Ø,145: PLOT 72,7: DRAW Ø,145
77Ø RETURN
779 REM get four pegs for a guess.
78Ø LET NG = 1
789 REM use subroutine to build display string for input prompt.
79Ø GO SUB 83Ø: INPUT (I$);G(NG): IF G(NG) < 1 OR G(NG) > 7 THEN GO TO 79Ø
799 REM backspace to remove last peg.
8ØØ IF G(NG) = 7 THEN LET NG = NG - 1: GO TO 79Ø
8Ø9 REM keep going till all four pegs are in and have been confirmed.
81Ø LET NG = NG + 1: IF NG < 6 THEN GO TO 79Ø
82Ø RETURN
829 REM make up string with colour codes and characters instead of numbers.
83Ø LET I$ = CHR$ 17 + CHR$ 7 + "YOUR GUESS "
84Ø IF NG <= 1 THEN LET NG = 1: RETURN
85Ø FOR J = 1 TO NG - 1: LET I$ = I$ + CHRS 16 + CHR$ G(J) + "A  ": NEXT J
86Ø RETURN
```

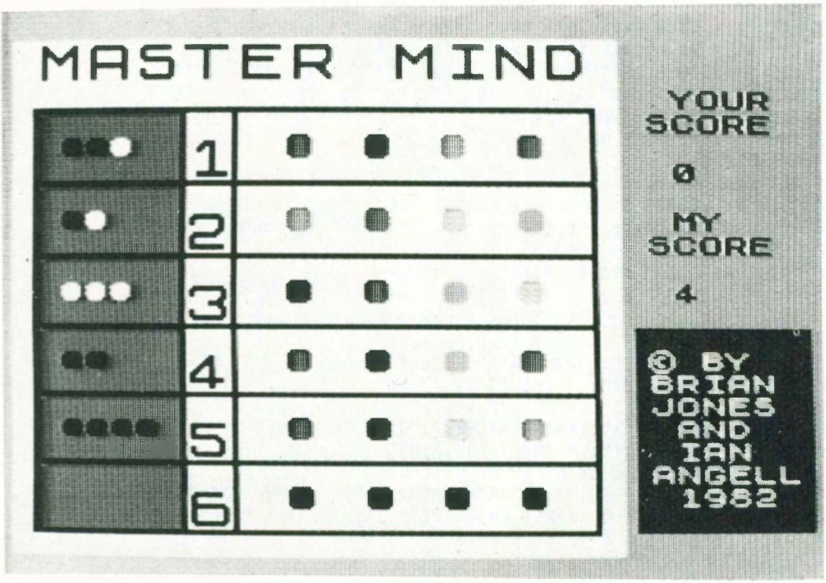


Figure 5.6

The routines taken from the 'CHARACTER GENERATOR' program have been left with their original numbering so that they can be easily identified and compared with the original. The display for the top part of the screen is built up in easy stages from characters, colours and lines so that the role of each stage of construction is straightforward and easily adjusted. The program uses also the techniques of dynamic INPUT strings (which will be discussed in chapter 13) to create the display for the lower part of the screen.

Exercise 5.7

Tidy up the 'MASTER MIND' program, giving it a structured appearance. Increase its legibility with suitable variable names especially for routines. Adapt the 'MASTER MIND' program to play against you. (Programs that do this have occasionally been published in the past in computing magazines. Some are still available; see Ahl, 1980 and Liffick, 1979).

Finally in this chapter, we give a simple illustration of how effective character graphics can be in producing a high-quality display. The following short program uses the 'chesspiece' characters from the cassette tape to display a chess board (for example, figure 5.7 – also see cover) and to move pieces in response to INPUTs. You can of course produce your own new chess set.

Listing 5.7

```
  9 REM set up to use just sets 1 and 5 for 16K use CLEAR 31830: S5 = 31575.
 10 CLEAR 64598: LET S5 = 64343: LET S1 = 15360
 20 INPUT " LOAD CHARS ? ";Y$: IF Y$ = "y"
    THEN LOAD "chesspiece" CODE S5 + 256,768

100 REM lay out board
110 INK 0: PAPER 0: BORDER 0: CLS
120 PAPER 6
129 REM print yellow strip around board.
130 FOR I = 1 TO 20: FOR J = 1 TO 2
140 PRINT AT 21 - I,J + 5;" ": PRINT AT I,J + 23;" "
150 PRINT AT J,I + 5;" ": PRINT AT J + 18,26 - I;" "
160 NEXT J: NEXT I
169 REM print squares of board.
170 LET C = 4
180 FOR I = 2 TO 8 STEP 2: FOR J = 2 TO 8 STEP 2
190 PAPER C: PRINT AT J + 1,I + 6;"  ": PRINT AT J + 2,I + 6;"  "
200 PRINT AT 19 - J,24 - I;"  ": PRINT AT 20 - J,24 - I;"  "
210 LET C = 7 - C: PAPER C
220 PRINT AT J + 1,24 - I;"  ": PRINT AT J + 2,24 - I;"  "
230 PRINT AT 19 - J,I + 6;"  ": PRINT AT 20 - J,I + 6;"  "
240 NEXT J: LET C = 7 - C: NEXT I
249 REM print letters and numbers around board.
250 PAPER 6
260 FOR I = 1 TO 8: PRINT AT 20,6 + 2*I;CHR$ (64 + I)
    : PRINT AT 2*I + 2,25;(9 - I): NEXT I

300 REM set out pieces
310 DIM B(8,8)
319 REM white pieces have ten added to the number identifying type of piece.
320 FOR I = 1 TO 8: LET B(2,I) = 6: LET B(7,I) = 16: NEXT I
330 FOR I = 1 TO 3: LET B(1,I) = 4 - I: B(1,I+5) = I
```

```
340 LET B(8,I) = 14 - I: LET B(8,I+5) = I + 10: NEXT I
350 FOR I = 4 TO 5: LET B(1,I) = 9 - I: LET B(8,I) = 19 - I: NEXT I
359 REM set pointers to routines.
360 LET move = 1000: LET input = 1100: LET piece = 1200: LET flash = 1300
    : LET charset = 1400: LET list = 1500
369 REM draw headings for move display.
370 PAPER 0: INK 7: PRINT AT 1,0;"WHITE": PRINT 1,27;"BLACK"
380 PLOT 0,159: DRAW 39,0: DRAW 0,-1: DRAW -39,0
390 PLOT 216,159: DRAW 39,0: DRAW 0,-1: DRAW -39,0
399 REM use transparent paper and then call routine to draw each piece.
400 PAPER 8: FOR K = 1 TO 2: FOR J = 1 TO 8: LET I = K: GO SUB piece
    : LET I = 9 - K: GO SUB piece : NEXT J: NEXT K

500 REM main program
509 REM array to hold moves and set no of moves to one.
510 DIM N$(2,100,5): BORDER 0: LET S = S1: GO SUB charset: LET N = 1
519 REM get whites move, flash squares specified.
520 LET I$ = "WHITE": GO SUB input: LET F = 1: GO SUB flash
529 REM allow move to be cancelled and re-entered.
530 INPUT "ACCEPT ? ";Y$: IF Y$ <> "Y" THEN LET F = 0: GO SUB flash: GO TO 520
539 REM move piece and display move on side.
540 GO SUB move: LET N$(1,N) = M$: GO SUB list
549 REM repeat above process for black side.
550 LET I$ = "BLACK": GO SUB input: LET F = 1: GO SUB flash
560 INPUT "ACCEPT ? ";Y$: IF Y$ <> "Y" THEN LET F = 0: GO SUB flash: GO TO 550
570 GO SUB move: LET N$(2,N) = M$: GO SUB list
579 REM next move.
580 LET N = N + 1: GO TO 520

1000 REM move
1009 REM move specified pieces and copy the pieces onto the screen.
1010 LET B(I2,J2) = B(I1,J1): LET I = I2: LET J =J2: GO SUB piece
1020 LET B(I1,J1) = 0: LET I = I1: LET J =J1: GO SUB piece
1030 RETURN

1100 REM input
1109 REM input coordinates of move square and destination square.
1110 INPUT (I$ + "'S MOVE No. " + STR$ N + " "); LINE M$;"-"; LINE T$
     : IF M$ = "STOP" OR M$ = " STOP " THEN STOP
1120 IF LEN M$ <> 2 OR LEN T$ <> 2 THEN GO TO input
1130 LET J1 = CODE M$(1) - 64: LET I1 = 9 - (VAL M$(2))
     : IF I1 < 1 OR I1 > 8 OR J1 < 1 OR J1 > 8 THEN GO TO input
1140 LET J2 = CODE T$(1) - 64: LET I2 = 9 - (VAL T$(2))
     : IF I2 < 1 OR I2 > 8 OR J2 < 1 OR J2 > 8 THEN GO TO input
1150 LET M$ = M$ + "-" + T$: RETURN

1200 REM piece
1201 REM IN: I,J
1209 REM redraw the square at I,J from the value stored in the board array.
1210 LET S = S5: GO SUB charset: LET B = B(I,J): INK 0
     : IF B > 10 THEN INK 7: LET B = B - 10
1220 LET B = 2*B: LET A$ = CHR$ (63 + B) + CHR$ (64 + B)
     : LET B$ = CHR$ (95 + B) + CHR$ (96 + B)
1230 IF B = 0 THEN LET A$ = "  ": LET B$ = A$
1240 PRINT AT I*2 + 1,J*2 + 6;A$: PRINT AT I*2 + 2,J*2 + 6;B$
1250 LET S = S1: GO SUB charset: RETURN

1300 REM flash
1301 REM IN: F,I1,J1,I2,J2
1309 REM change the flash attribute of two squares I1,J1 and I2,J2.
1310 FLASH F: OVER 1: INK 8: PRINT AT I2*2 + 1,J2*2 + 6;"  "
     : PRINT AT I2*2 + 2,J2*2 + 6;"  "
1320 PRINT AT I1*2 + 1,J1*2 + 6;"  " : PRINT AT I1*2 + 2,J1*2 + 6;"  "
1330 FLASH 0: OVER 0: RETURN
```

```
1400 REM charset
1401 REM IN: S
1409 REM change CHARS to S to use alternate sets.
1410 LET HI = INT (S/256): LET LO = S - 256*HI
1420 POKE 23606,LO: POKE 23607,HI: RETURN

1500 REM list
1509 REM list last 16 moves of each side on screen.
1510 LET T = N: IF T < 16 THEN LET T = 16
1520 FOR I = T - 15 TO T: PRINT AT I - T + 18,0; INK 7;N$(1,I)
1530 PRINT AT I - T + 18,27; INK 7;N$(2,I): NEXT I
1540 RETURN
```

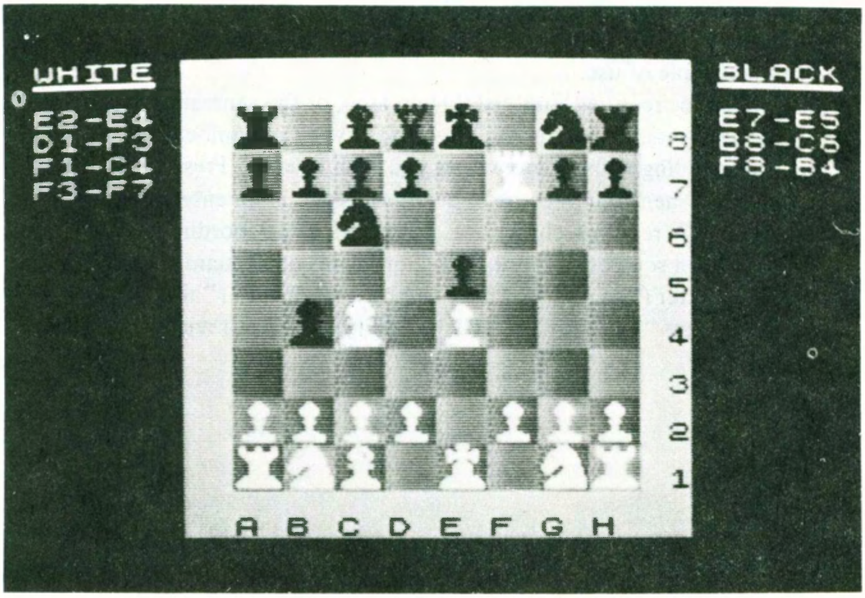


Figure 5.7

Exercise 5.8

Adapt the program in listing 5.7 to check for illegal moves and add facilities for castling and *en passant* captures. If you have a *lot of time!!* to spare then add routines to make the computer play against you (see Liffick, 1979).

In the next chapter we shall consider how character graphics and our knowledge of two-dimensional geometry may be combined to form data displays.

Complete Programs

I. Listing 5.1. Data required: A$. A$ should be any non-graphics mode character, try "X" or "x".

II. Listing 5.2 ('main program' and 'big pixels'). Data required: ROW and COL where $0 \leqslant \text{ROW} \leqslant 16$ and $0 \leqslant \text{COL} \leqslant 24$. Try (0, 0) and (1, 1).

III. Listing 5.3. Data required: U\$ and eight BINary numbers, each at most eight bits long to redefine the graphics character specified by U\$. Try "A" and 111, 111000, 1010101, 101, 11111001, 1001, 1111, 10110; and "B", "C" and "D", each with permutations of these eight numbers. Then press keys "A" to "U" to display the equivalent graphics characters or press ENTER to restart program.

IV. Listing 5.4: no data required. Program III must be used to create graphics characters "**A**", "**B**", "**C**" and "**D**".

V. Listing 5.5: the CHARACTER GENERATOR. Read text for description and example of use.

VI. Listing 5.6: requires 'masterset' from tape, or the generation of your own character sets. To play MASTER MIND, type the number ("1" to "6") corresponding to the colour of peg you wish to enter. Pressing "7" removes last peg. When four pegs are in position, type "1" to enter guess.

VII. Listing 5.7: requires 'chesspiece' from tape. Type coordinates of start and destination squares of move. The coordinates of a square are given by a capital letter ("A" to "H") followed by a number ("1" to "8"). Try "E2" followed by "E4". To ACCEPT MOVE type "Y", to reject the move type "N".

6 *Diagrams and Data Graphs*

More information is available to more people than ever before. Businessmen are being overwhelmed by massive documents containing reams of statistics on every subject from capital expenditure to market research. Sociologists bombard us with figures on child development and the increasing percentage of octogenarians in Bournemouth. Worst of all, computers are piling up printouts of dreary data covering every topic from astrology to zoology. Obviously something must be done! Computers have helped to create the problem and they can also help to solve it. The data must be presented in a more digestible manner: as pie-charts, histograms, scientific graphs or just plain diagrams. With the advent of desktop computers the increasing sales of programs that produce these displays has made this one of the major growth areas in computer graphics. In this chapter we shall see how such diagrams can be constructed with ease, given just a few tools to aid our draughtsmanship.

Cursors

Before us is a sheet of PAPER on which we shall place objects. We require some method for accurately controlling the position of these objects. This is usually achieved with a cursor, which may be externally controlled by a joystick or other analogue input devices. We, however, will use the keyboard to control movements. This may not be as convenient to use as a joystick, but it requires no extra expenditure on peripherals, which would achieve only the same effect anyway. The 'cursor' routine, listing 6.1, produces a pair of crosswires that are OVERlayed on the display using transparent INK (INK 8). These crosswires can specify the pixel at their intersection or the character block in which they cross. The cursor is moved in any one of eight directions by the keys around the "F" (see figure 6.1). If you have a joystick or similar peripheral attached to your computer, then alter the 'cursor' routine so that it receives information from that rather than the keyboard.

Pressing "F" itself centres the cursor on the screen. When a key is held down the speed of movement gradually increases: a cursor that always moves just one pixel per key depression is tedious to use! To aid in positioning the cursor there is a Lattice that is switched on and off by pressing "L". This is automatically removed, if it is on, when you press ENTER to enter a point.

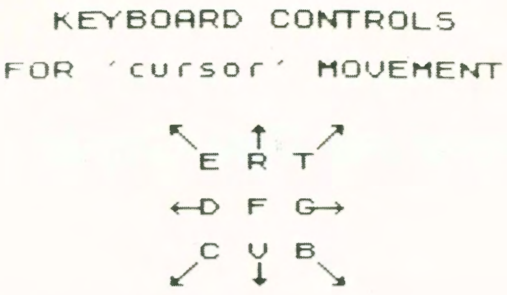


Figure 6.1

Listing 6.1

```
5700 REM cursor
5701 REM IN: PX,PY
5702 REM OUT: PX,PY,ROW,COL
5709 REM the next line is only executed on the first call to cursor.
5710 LET PX = 128: LET PY = 88: LET cursor = 5720: LET grid = 5900
5719 REM start of main cursor routine.
5720 INK 8: LET A = 1: LET FLAG = 1: OVER 1
5729 REM wait for keyboard to be clear before looking for commands.
5730 IF INKEY$ <> "" THEN GO TO 5730
5740 PLOT PX,0: DRAW 0,175: PLOT 0,PY : DRAW 255,0
5750 LET A$ = INKEY$: IF A$ = "" THEN LET A = 1: GO TO 5750
5760 IF CODE A$ > 64 AND CODE A$ < 96 THEN LET A$ = CHR$ (CODE A$ + 32)
5770 PLOT PX,0: DRAW 0,175: PLOT 0,PY : DRAW 255,0
5780 IF (A$ = "e" OR A$ = "d" OR A$ = "c") AND PX >= A THEN LET PX = PX - A
5790 IF (A$ = "c" OR A$ = "v" OR A$ = "b") AND PY >= A THEN LET PY = PY - A
5800 IF (A$ = "e" OR A$ = "r" OR A$ = "t") AND PX <= 175-A THEN LET PY = PY + A
5810 IF (A$ = "t" OR A$ = "g" OR A$ = "b") AND PX <= 255-A THEN LET PX = PX + A
5820 IF A$ = "f" THEN LET PX = 128: LET PY = 88
5830 IF A$ = "l" THEN GO SUB grid
5840 IF A$ <> CHR$ 13 THEN LET A = A + 1: GO TO 5740
5850 LET COL = INT (PX/8): LET ROW = INT ((175-PY)/8)
5860 IF FLAG = -1 THEN GO SUB grid
5870 INK 0: OVER 0
5880 RETURN

5900 REM grid
5909 REM routine changes flag to remember whether grid is showing on screen.
5910 FOR J = 0 TO 255 STEP 8: PLOT J,0: DRAW 0,175: IF J < 176
     THEN PLOT 0,J: DRAW 255,0
5920 NEXT J: LET FLAG = -FLAG: PLOT 255,0: DRAW 0,175: DRAW -255,0
5930 RETURN
```

Exercise 6.1

Write a 'main program' that calls 'cursor' and then prints a coloured square on the screen at the specified block. It should also print out the ROW/COLumn position of the block. Change the 'cursor' routine so that the standard cursor keys ("5", "6", "7" and "8") can be used to move our cursor in character block steps about the graphics area.

Attributes

We can extend the idea from exercise 6.1 to produce routines that change the attributes of a given block or group of blocks without affecting their contents in the display file. Listing 6.2 shows two routines that do this for the PAPER and INK colours of a specified set of blocks.

Listing 6.2

```
4000 REM paper
4010 GO SUB cursor: INPUT "WHAT COLOUR ";P: INPUT "No. OF BLOCKS (ROW*COL) "
     ;R;"*";C
4020 FOR I = ROW TO ROW + R - 1: FOR J = COL TO COL + C - 1
4030 PRINT AT I,J; PAPER P; INK 8; OVER 1;" ": NEXT J: NEXT I
4040 RETURN

4100 REM ink
4110 GO SUB cursor: INPUT "WHAT COLOUR ";P: INPUT "No. OF BLOCKS (ROW*COL) "
     ;R;"*";C
4120 FOR I = ROW TO ROW + R - 1: FOR J = COL TO COL + C - 1
4130 PRINT AT I,J; PAPER 8; INK P; OVER 1;" ": NEXT J: NEXT I
4140 RETURN
```

Exercise 6.2
Write a 'main program' that LISTs part of itself and then allows you to highlight parts of the listing using different INK or PAPER colours. Write routines to be placed at lines 4050 and 4150, which alter the FLASH or BRIGHT attributes of blocks. (Why will this mean adding FLASH 8 and BRIGHT 8 to the 'paper' and 'ink' routines?). Combine all four routines into one named 'attribute'.

Points and Lines

Having achieved simple interactive control over the appearance of our diagram, we must now fill in the details of the picture. This requires routines for PLOTting pixel points on the screen and for DRAWing lines. We could use program

Listing 6.3

```
4200 REM point
4210 INPUT "PRESS ENTER FOR CURSOR"; LINE A$
4220 GO SUB cursor: INPUT "OVER ( 1 OR 0 ) ";O: INPUT "COLOUR ";C
4230 PLOT INK C; OVER O;PX,PY
4240 RETURN

4300 REM line
4310 INPUT "PRESS ENTER FOR CURSOR"; LINE A$
4320 GO SUB cursor: LET SX = PX: LET SY = PY
4330 INPUT "PRESS ENTER FOR CURSOR"; LINE A$
4340 GO SUB cursor: INPUT "OVER ( 1 OR 0 ) ";O: INPUT "COLOUR ";C
4350 PLOT PAPER 8; INK C; OVER O;SX,SY
4360 DRAW PAPER 8; INK C; OVER O;PX - SX,PY - SY
4370 RETURN
```

listing 1.8, but it would be boring to go over every pixel in a line. Instead we use two routines 'point', which PLOTs individual pixels, and 'line', in which we simply specify the two end points of a line before joining them up. These two routines are shown in listing 6.3.

Exercise 6.3
Write a 'main program' that allows you to use 'point' and 'line' to sketch a picture of your Spectrum. Incorporate 'circle1' or 'circle2' from listing 2.10 in a 'circle' routine; where necessary, parameters, the centre and radius, are added using the 'cursor'. Adapt your program from exercise 1.3 for use as a routine to draw an *n*-sided polygon using the 'cursor' to enter the points. Add an extra option to the 'line' routine to allow DRAWing of curved lines and then construct a diagram similar to figure 1.10a (it will be easier to use eight points around the edge rather than twelve).

If you have a graphics pad then you can copy rough sketches from the pad into the machine using an adjusted version of 'line' and 'point'. You should then write programs to tidy up these pictures; that is, straighten lines and smooth out curves.

Save and Load

Having spent some time and effort drawing and colouring a pretty picture, we might wish to SAVE it for future reference. Another essential requirement is to be able to LOAD a picture drawn by another program, and then alter it. The two routines 'save' and 'load' that make this possible are shown in listing 6.4.

Listing 6.4

```
4500 REM save
4510 INPUT "NAME OF PICTURE ? ";N$: IF N$ = "" THEN RETURN
4520 SAVE (N$)SCREEN$
4530 RETURN

4600 REM load
4610 INPUT "NAME OF PICTURE ? ";N$: IF N$ = "" THEN RETURN
4620 LOAD (N$)SCREEN$
4630 RETURN
```

Exercise 6.4
Run the program in listing 3.1 (or LOAD figure 3.1 if you have it stored) and then change the display to green INK on black PAPER. Furthermore, there are similar statements in both routines 'save' and 'load'. To save space, combine 'save' and 'load' into a single routine.

Labels

Listings 6.1 to 6.4 form the basis of our diagram construction package, but as yet we have not put any labels on the diagrams. This is achieved with the 'label' routine, listing 6.5. Note that in this routine we make references to routines not yet written. This is a typical situation given the structured modular approach adopted throughout this book. In practice this means that whenever we come up against a problem that is too large to tackle immediately, we simply give it a name and deal with it later. In 'label' we decide that character strings drawn using symbols from set 5 will be printed vertically. For reasons explained later, we also define a special prefix (7:) that indicates that the string of characters for a label is to be printed in narrow characters using a special routine 'thin'. We also assume that there is a routine named 'set' that enables us to switch between character sets.

Listing 6.5

```
5400 REM label
5410 GO SUB cursor: INPUT "SET:STRING ";A$: IF A$ = "" THEN GO TO 5410
5420 INPUT "COLOUR ";C$: INK VAL("0" + C$)
5430 IF A$( TO 2) = "1:" THEN PRINT  AT ROW,COL;A$(3 TO ): GO TO 5510
5439 REM check for special set numbers, set 5 is printed vertically.
5440 IF A$( TO 2) <> "5:" THEN GO TO 5470
5450 LET S = 5: GO SUB set: LET A$ = A$(3 TO )
5460 FOR I = 1 TO LEN A$: PRINT AT ROW-I+1,COL;A$(I): NEXT I: GO TO 5510
5469 REM set 7 specifies that the thin routine should be used for printing.
5470 IF A$( TO 2) <> "7:" THEN GO TO 5490
5480 LET A$ = A$(3 TO ): GO SUB thin: GO TO 5510
5489 REM check valid number is given.
5490 LET S = VAL A$(1): IF S > 6 OR S < 1 OR A$(2) <> ":" THEN GO TO 5510
5499 REM print out string in appropriate set.
5500 GO SUB set: PRINT AT ROW,COL;A$(3 TO )
5510 LET S = 1: GO SUB set
5520 INK 0: RETURN
```

Data Graphs

In an attempt to tidy up some of the loose ends, we introduce a general purpose 'main program' and a 'query' routine in listing 6.6. Inevitably this creates as many new loose ends as it ties up, but it should help to give us an overall view of the tasks left unsolved. We introduce identifiers (lower case) for routines, to be written later that, among other things, will allow us to draw special types of diagrams; for example, 'histo'grams, 'pie'-charts and 'graph's.

The section of listing 6.6 before the 'main program' is a cleaned-up version of those parts of the CHARACTER GENERATOR from chapter 5 necessary for using alternate character sets.

A program containing all the routines from listings 6.1 to 6.7 was used to draw most of the diagrams in this book that were not directly produced by example programs. It was used also for adding labels to many of the figures that were produced by programs; for example, figure 4.1.

Listing 6.6

```
  8 REM for 16K machines make changes as given in listing 5.5.
  9 REM set up to use alternate character sets.
 10 CLEAR 62294
 20 INK 0: PAPER 7: CLS: DIM S(6)
 30 FOR I = 1 TO 6: READ S(I): NEXT I: DATA 15360,62039,62807,63575
    ,64343,64848
 40 LET set = 50: LET S = 1: GO SUB set: GO TO 200
 50 REM set/change to set S
 60 LET HI = INT (S(S)/256): LET LO = S(S) - HI*256
 70 POKE 23606,LO: POKE 23607,HI: RETURN
 80 REM charload
 90 INPUT "WHAT FILE NAME ? ";N$: IF N$ = "" THEN RETURN
100 INPUT "LOAD WHICH SET ? ";S: IF (S<2 OR S>6) AND S<>11 THEN GO TO 80
110 INPUT "Start tape, then press enter."; LINE X$
120 IF S = 11 THEN LOAD N$ CODE (S(5) + 256),768 + 168: RETURN
130 IF S = 6 THEN LOAD N$ CODE (S(S) + 512),168: RETURN
140 LOAD N$ CODE (S(S) + 256),768: RETURN

200 REM main program
210 LET restart = 280: LET query = 500
220 LET histo = 1000: LET pie = 2000: LET graph = 3000
230 LET paper = 4000: LET ink = 4100: LET point = 4200: LET line = 4300
    : LET save = 4500: LET load = 4600: LET number = 4700
240 LET charload = 80: LET create = 5000: LET thin = 5200
250 LET label = 5400: LET cursor = 5700
260 LET I$ = "DEFINE CHARACTERS ? ": GO SUB query: IF YES THEN GO SUB create
270 CLS: LET I$ = "LOAD CHARACTERS ? ": GO SUB query: IF YES THEN GO SUB charload
    : GO TO 270
280 LET I$ = "LOAD PICTURE ? ": GO SUB query: IF YES THEN GO SUB load
290 LET I$ = "DRAW DIAGRAM ? ": GO SUB query: IF NOT YES THEN GO TO 360
300 LET I$ = "HISTOGRAM, PIE-CHART OR GRAPH ? ": GO SUB query
310 LET diagram = 0: IF A$ = "h" THEN LET diagram = histo
320 IF A$ = "p" THEN LET diagram = pie
330 IF A$ = "g" THEN LET diagram = graph
340 IF diagram = 0 THEN GO TO 280
350 GO SUB diagram
360 LET I$ = "LABEL PICTURE ? ": GO SUB query: IF YES THEN GO SUB label
    : GO TO 360
370 LET I$ = "COLOUR PAPER ? ": GO SUB query: IF YES THEN GO SUB paper
    : GO TO 370
380 LET I$ = "COLOUR INK ? ": GO SUB query: IF YES THEN GO SUB ink
    : GO TO 380
390 LET I$ = "DRAW POINT ? ": GO SUB query: IF YES THEN GO SUB point
    : GO TO 390
400 LET I$ = "DRAW LINE ? ": GO SUB query: IF YES THEN GO SUB line
    : GO TO 400
410 LET I$ = "END PICTURE ? ": GO SUB query: IF NOT YES THEN GO TO restart
420 LET I$ = "SAVE PICTURE ? ": GO SUB query: IF YES THEN GO SUB save
430 STOP

500 REM query
501 REM IN  : I$
502 REM OUT : A$,YES
510 LET YES = 0
520 INPUT (I$); LINE A$: IF A$ = "" THEN RETURN
530 LET A$ = A$(1): IF CODE A$ < 96 THEN LET A$ = CHR$ (CODE A$ + 32)
540 LET YES = (A$ = "y")
550 RETURN
```

Special Characters

To complete our preparations for the 'label' routine, we need to create a set of ROTATED characters, which is placed in set 5 for use in writing vertical labels. This is done in the routine 'create' (listing 6.7), which uses some of the techniques from the CHARACTER GENERATOR to copy a rotated version of set 1 into set 5.

Listing 6.7

```
5000 REM create/characters
5009 REM create characters for histogram routine.
5010 DIM P(8): DIM D(3): LET D(1) = 15: LET D(2) = 255: LET D(3) = 240
5020 FOR I = 0 TO 6: FOR J = 0 TO 7
5030 LET P(1) = USR "A" + I*8 + J: LET P(2) = USR "H" + I*8 + J
5040 LET P(3) = USR "O" + I*8 + J
5050 FOR K = 1 TO 3: POKE P(K),0: NEXT K
5060 IF J > I THEN FOR K = 1 TO 3: POKE P(K),D(K): NEXT K
5070 NEXT J: NEXT I
5078 REM copy set 1 to set 5 with a rotation to get sideways characters.
5079 REM for 16K machines Q = 31831.
5080 DIM B$(8,8): LET T = 256: FOR I = 1 TO 8: LET T = T/2: LET P(I) = T
     : NEXT I: LET P = 15616: LET Q = 64599
5090 FOR I = 32 TO 127: FOR J = 1 TO 8: LET B$(J) = "00000000": NEXT J
5100 FOR J = 1 TO 8: LET T = PEEK P: FOR K = 1 TO 8
5110 IF T >= P(K) THEN LET T = T - P(K): LET B$(9-K,J) = "1"
5120 NEXT K: LET P = P + 1: NEXT J
5130 FOR J = 1 TO 8: POKE Q,VAL("BIN " + B$(J)): LET Q = Q + 1: NEXT J
5140 PRINT AT 10,10;"ROTATING ";CHR$ I
5150 NEXT I: CLS
5160 RETURN

5200 REM thin
5201 REM IN  : ROW,COL,A$
5209 REM print every other character in left half of blocks.
5210 LET S = 4: GO SUB set: OVER 1: LET TC = COL
5220 FOR I = 1 TO LEN A$ STEP 2: PRINT AT ROW,TC;A$(I): LET TC = TC + 1: NEXT I
5229 REM print other characters in between in right half of blocks.
5230 LET S = 3: GO SUB set: LET TC = COL
5240 FOR I = 2 TO LEN A$ STEP 2: PRINT AT ROW,TC;A$(I): LET TC = TC + 1: NEXT I
5250 LET S = 1: GO SUB set: OVER 0
5260 RETURN
```

Before generating the rotated characters, the 'create' routine initially redefines the user-defined graphics set as three sets of seven characters, see figure 6.2, that allow either the left or right halves, or all of a character block, to be INKed in from the bottom up. These characters may be used, by OVERprinting, to obtain horizontal bars of varying thicknesses on the screen; but they are primarily for use in the 'histo'gram routines that follow.

For the remainder of this chapter we shall discuss some common types of data display. The vast number of variations on each of the major themes of graphical display make it impractical to discuss all possibilities. We shall, therefore, consider in detail just a few examples from each of the three main types of display: histograms, pie-charts and graphs. From these simple examples it should

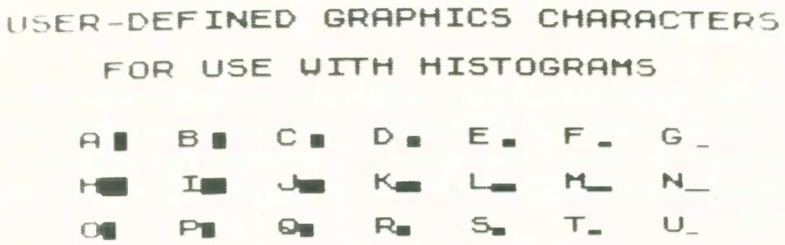


Figure 6.2

be possible to construct variations on these routines to display data in any manner required. Again note the changes for the 16K machine given in appendix A.

Histograms

Histograms (or bar-charts) can be constructed by our programs to any height in pixels and in any colour, provided both the width of the bars and the separation between them are at least half a character block. Since this is quite normal for histograms anyway, we can formulate a method for calculating the spacing and width of bars once we know how many there are. The first part of the 'histo' routine (listing 6.8) draws the axes, labels the vertical and then asks how many bars are required. The width of the bars (X), and the size of the GAPs between the bars, are both calculated (in multiples of half blocks) by a method that ensures that $1/2 \leqslant \text{GAP} \leqslant \text{X}$. On receiving each data item, the routine uses the full and half-width characters together with the user-defined characters (from 'create') to build up strings representing the scale height of the bar. These strings are used to fill in any full or half-width character blocks assigned to that bar by prefixing them with '5:' and using 'bar' to print them vertically. Note that the axes lie in the character block just outside the area containing the bar-chart to ensure their independence from any colour changes made inside this area.

Figure 6.3, a diagram presenting the annual rainfall in Egham, was constructed using 'histo' (/type1) from listing 6.8 and then 'label'ed.

Exercise 6.5

As variations on the standard 'histo' routine we can write replacement routines that can be MERGEd as required. Write a routine that calculates the position for pairs of bars, where the bars within a pair are separated by half a block, but pairs are separated from neighbouring pairs by at least one block. Use this to construct diagrams similar to figure 6.4.

An example of a replacement 'histo' routine is given in listing 6.9. In this version of 'histo' (/type2) the calculation of the width of the bars is altered to provide whole character block values for X and GAP. Two data values are requested for each bar, specifying the MAXimum and MINimum height range of

Listing 6.8

```
1000 REM histo/type1
1009 REM set pointer to subroutine which prints bars.
1010 LET bar = 1320
1019 REM find scale and draw axes.
1020 INPUT "RANGE OF VERTICAL ";YB;" TO ";YT
1030 IF YB >= YT THEN GO TO 1020
1040 LET YSCALE = 128/(YT - YB)
1050 PLOT 47,152: DRAW 0,-129: DRAW 201,0
1059 REM label vertical axis.
1060 LET YDIF = (YT - YB)/4: LET TICK = YB
1070 FOR I = 1 TO 5: LET TK = INT (TICK + 0.5)
1080 LET Y = 32*I - 8: PLOT 47,Y: DRAW -3,0: LET ROW = INT ((176 - Y)/8)-1
1090 LET A$ = STR$ TK: IF LEN A$ > 3 THEN LET A$ = A$( TO 3)
     : IF TK > 999 OR TK < -99 THEN LET A$ = "***"
1100 LET COL = 1 : IF TK >= 0 THEN LET COL = COL + 1
1110 IF ABS TK < 10 THEN LET COL = COL + 1
1120 IF ABS TK < 100 THEN LET COL = COL + 1
1130 PRINT AT ROW,COL;A$: LET TICK = TICK + YDIF
1140 NEXT I
1149 REM find number of bars required and calculate how they can fit in.
1150 INPUT "No. OF BARS ";NB
1160 LET X = INT (25/NB + 0.5)/2: LET GAP = INT (50/NB - 2*X)/2
1170 LET GP = (25 - NB*X - (NB - 1)*GAP)*2: LET GP = INT ((GP + 1)/2)/2
1179 REM position column pointer at first bar then repeat loop for each bar.
1180 LET COL = 6 + GP: FOR I = 1 TO NB
1190 LET I$ = "DATA FOR BAR " + STR$ I + " ": INPUT (I$);D;" COLOUR ";C
     : INK C: LET W = X
1199 REM values below the horizontal axis are treated as single pixel height.
1200 IF D <= YB THEN LET D = 0: LET H = 1: GO TO 1220
1209 REM calculate the number of whole blocks of height for bar.
1210 LET D = D - YB: LET H = INT (D*YSCALE + 0.5)
1220 LET IH = INT (H/8): LET L$ = "": LET M$ = "": LET R$ = ""
1229 REM construct strings to height for left, middle and right of bars.
1230 FOR J = 1 TO IH: LET L$ = L$ + CHR$ 133: LET M$ = M$ + CHR$ 143
     : LET R$ = R$ + CHR$ 138
1239 REM find number of pixels of height for special character at top.
1240 NEXT J: LET RH = H - IH*8: IF RH = 0 THEN GO TO 1260
1249 REM add special characters to tops of bars.
1250 LET L$ = L$ + CHR$ (151 - RH): LET M$ = M$ + CHR$ (158 - RH)
     : LET R$ = R$ + CHR$ (165 -RH)
1259 REM  if column pointer is on a half-block print right biased part of bar.
1260 IF COL = INT (COL) THEN GO TO 1280
1270 LET A$ = L$: GO SUB bar: LET W = W - 0.5: LET COL = COL + 0.5
1279 REM print as many whole block bars as needed.
1280 IF W >= 1 THEN LET A$ = M$: GO SUB bar: LET W = W - 1
     : LET COL = COL + 1: GO TO 1280
1289 REM see if left biased half-width bar is needed.
1290 IF W > 0.1 THEN LET A$ = R$: GO SUB bar: LET COL = COL + 0.5
1299 REM move column pointer to next bar and re-do loop.
1300 LET COL = COL + GAP: NEXT I
1310 INK 0: RETURN

1320 REM bar
1321 REM IN  : COL,A$
1329 REM print A$ vertically from row 19 upwards in COLumn.
1330 FOR K = 1 TO LEN A$: PRINT AT 19 - K, INT COL;A$(K): NEXT K
1340 RETURN
```

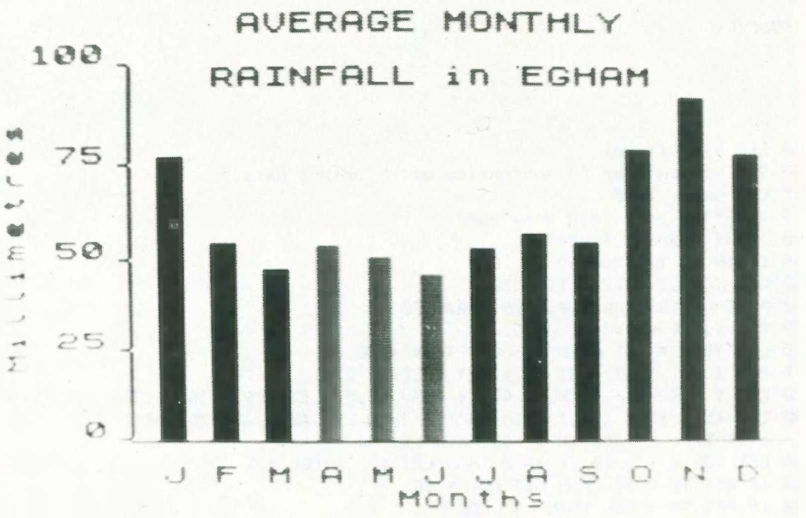


Figure 6.3

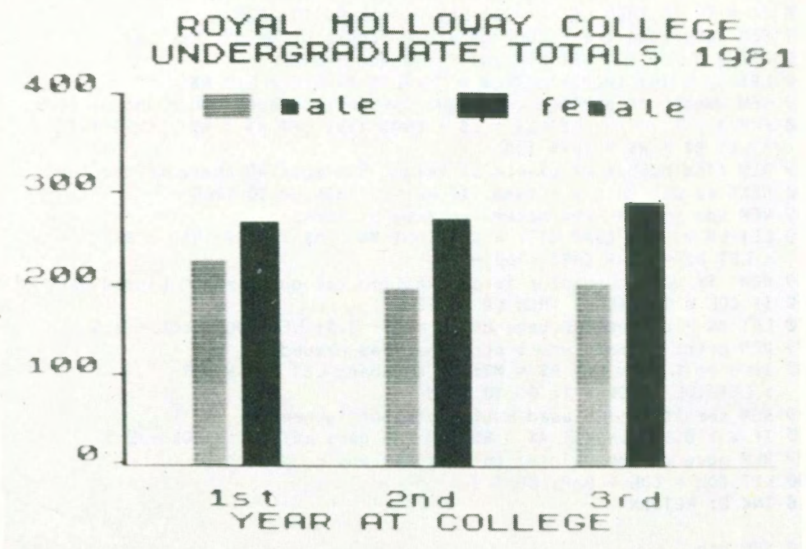


Figure 6.4

the bar. By simply OVERprinting the bars, charts like figure 6.5, of the monthly temperature variation in Egham, can be produced. In order to understand this fully, as well as other programs in the book, it is a good idea to scatter PRINT statements throughout the listings, so you can follow the logic of the algorithms as they are executed. A very nice feature in BASIC is that you can add extra

Listing 6.9

```
1000 REM histo/type2
1009 REM set pointer to subroutine which prints bars.
1010 LET bar = 1320: DIM O$(2,3): LET O$(1) = "MAX": LET O$(2) = "MIN"
1019 REM find scale and draw axes.
1020 INPUT "RANGE OF VERTICAL ";YB;" TO ";YT
1030 IF YB >= YT THEN GO TO 1020
1040 LET YSCALE = 128/(YT - YB)
1050 PLOT 47,152: DRAW 0,-129: DRAW 201,0
1059 REM label vertical axis.
1060 LET YDIF = (YT - YB)/4: LET TICK = YB
1070 FOR I = 1 TO 5: LET TK = INT (TICK + 0.5)
1080 LET Y = 32*I - 8: PLOT 47,Y: DRAW -3,0: LET ROW = INT ((176 - Y)/8)-1
1090 LET A$ = STR$ TK: IF LEN A$ > 3 THEN LET A$ = A$( TO 3)
     : IF TK > 999 OR TK < -99 THEN LET A$ = "***"
1100 LET COL = 1 : IF TK >= 0 THEN LET COL = COL + 1
1110 IF ABS TK < 10 THEN LET COL = COL + 1
1120 IF ABS TK < 100 THEN LET COL = COL + 1
1130 PRINT AT ROW,COL;A$: LET TICK = TICK + YDIF
1140 NEXT I
1149 REM find number of bars required and calculate an integer width for them.
1150 INPUT "No. OF BARS ";NB: OVER 1
1160 LET GAP = INT (13/NB): LET X = INT ((26 - GAP*NB)/NB)
1170 LET GP = (25 - NB*X - (NB - 1)*GAP): LET GP = INT ((GP + 1)/2)
1179 REM position column pointer at first bar then repeat loop for each bar.
1180 LET COL = 6 + GP: FOR I = 1 TO NB
     : INPUT ("COLOUR FOR BAR " + STR$ I + ": ");C: INK C
1189 REM loop to input maximum and minimum values for bar.
1190 FOR O = 1 TO 2: LET I$ = "DATA " + O$(O) + " FOR BAR " + STR$ I + " "
     : INPUT (I$);D: LET W = X
1200 IF D <= YB THEN LET D = 0: LET H = 1: GO TO 1220
1209 REM calculate the number of whole blocks of height for bar.
1210 LET D = D - YB: LET H = INT (D*YSCALE + 0.5)
1220 LET IH = INT (H/8): LET M$ = ""
1229 REM construct string to height for bar.
1230 FOR J = 1 TO IH: LET M$ = M$ + CHR$ 143
1239 REM add special character for remaining height.
1240 NEXT J: LET RH = H - IH*8: IF RH = 0 THEN GO TO 1260
1250 LET M$ = M$ + CHR$ (158 - RH)
1259 REM start at same column for both bars in pair.
1260 IF O = 1 THEN LET OCOL = COL
1270 IF O = 2 THEN LET COL = OCOL
1279 REM output required number of whole blobk width bars.
1280 IF W >= 1 THEN LET A$ = M$: GO SUB bar: LET W = W - 1
     : LET COL = COL + 1: GO TO 1280
1290 LET COL = COL + GAP: NEXT O: NEXT I
1300 INK 0: OVER 0: RETURN

1320 REM bar
1321 REM IN  : COL,A$
1329 REM print A$ vertically from row 19 upwards in COLumn.
1330 FOR K = 1 TO LEN A$: PRINT AT 19 - K, INT COL;A$(K): NEXT K
1340 RETURN
```

STOPs to a program, interrogate the variable values, and then CONTinue, without affecting the status of the program. This is a very useful tool for debugging a program, as well as an aid to understanding a listing.

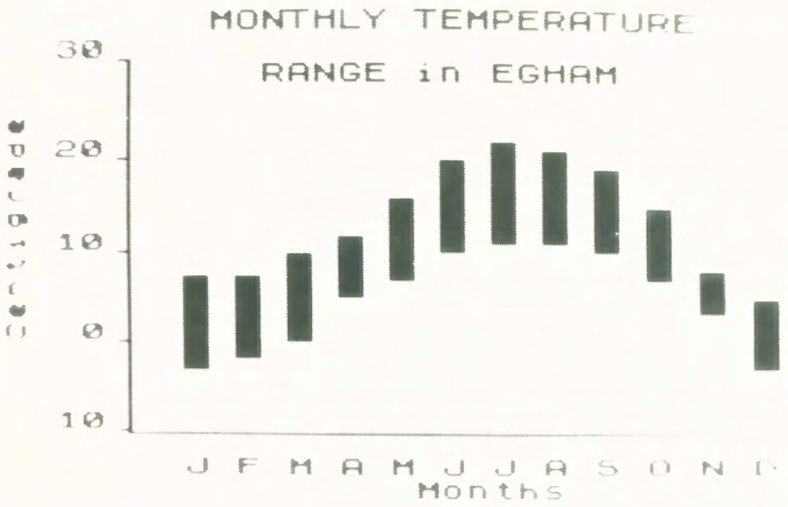


Figure 6.5

Exercise 6.6
Try using a different colour PAPER when OVERprinting the bars to produce two-coloured bars. Be careful when using data values giving scale heights with less than eight pixels difference!

Problems will occur when using two different colours for one bar (exercise 6.6). If the MAXimum and MINimum values both fall in the same character block, then three colours, these two plus the background, will be required within one block. We may check whether problems will occur by comparing the number of character blocks in the string produced for the MINimum data with that for the MAXimum data. If they are of equal length then the simplest way of avoiding trouble is to truncate the MINimum string by removing the last character. This will remove any problems with the colours on the display but will be slightly inaccurate. This whole problem will be avoided if we make the colour of the lower part of the bar the same as the background colour, thus ensuring that the bar is accurately drawn with only two different colours in each block.

There are many, many more variations possible; for example, drawing bars above and below a central line in order to display fluctuations in currency exchange rates. The fundamental ideas we have introduced should enable you to produce any histogram to your own specification.

Pie-Charts

The pie-chart is a favourite with economists and biologists who delight in telling us how big each slice of our capital expenditure cake is, or alternatively which bacteria are eating it. The usual requirements of a pie-chart program are that it should draw pies of variable radii, it must be possible to pull out slices of the pie from the centre, and provision for filling in or cross-hatching these slices must be made available. The 'pie'-chart routine given in listing 6.10 achieves the first two

Listing 6.10

```
2000 REM pie/chart
2009 REM set pointers to routines.
2010 LET hatch = 2300: LET in = 2800
2019 REM all segments are input and totalled to calculate angular scale.
2020 INPUT "No. OF SEGMENTS ";NB: INPUT "COLOUR ";C$: LET C = VAL ("0" + C$)
     : LET TOT = 0
2030 DIM D(NB): FOR I = 1 TO NB: INPUT ("DATA " + STR$ I + ": ");D(I)
     : LET TOT = TOT + D(I): NEXT I
2039 REM use cursor to specify centre of pie.
2040 INPUT "PRESS ENTER FOR CENTERING PIE";LINE Y$
2050 GO SUB cursor: LET XC = PX: LET YC = PY
2060 INPUT "RADIUS (IN PIXELS) ";RAD
2070 LET ASCALE = 2*PI/TOT: LET A1 = PI/2
2079 REM any wedge may be pulled out by moving the cursor away from the centre.
2080 FOR I = 1 TO NB
2090 INPUT "PRESS ENTER FOR CENTERING WEDGE";LINE Y$
2100 LET PX = XC: LET PY = YC: GO SUB cursor: LET ANG = ASCALE*D(I)
     : LET A2 = A1 - ANG
2109 REM if wedge is to move out calculate displacement along it's bisector.
2110 IF PX = XC AND PY =YC THEN GO TO 2140
2120 LET A3 = A1 - ANG/2: LET DIST = SQR ((PX - XC)*(PX - XC) +
     (PY - YC)*(PY - YC))
2130 LET PX = INT (XC + DIST*COS A3 + 0.5): LET PY = INT (YC + DIST*SIN A3
     + 0.5)
2139 REM enquire whether hatching is required and what type.
2140 INPUT "HATCH (x,y,b,n) ";H$: IF CODE H$ < 96
     THEN LET H$ = CHR$ (CODE H$ + 32)
2149 REM if hatching is wanted input gap size between lines and offset.
2150 IF H$ <> "n" THEN INPUT "JUMP ";JUMP,"REM ";REM
2159 REM draw segment of pie.
2160 INK C: PLOT PX,PY: LET X1 = INT (RAD*COS A1 + 0.5)
     : LET Y1 = INT (RAD*SIN A1 + 0.5): DRAW X1,Y1
2170 LET A2S = A2: LET ASTO = ANG: LET XA = PX + X1: LET YA = PY + Y1
2180 LET XB = INT (RAD*COS A2 + 0.5): LET YB = INT (RAD*SIN A2 + 0.5)
2190 FOR T = A1 TO A2 STEP -3/RAD
2200 LET X2 = INT (RAD*COS T + 0.5): LET Y2 = INT (RAD*SIN T + 0.5)
     : DRAW X2 - X1,Y2 - Y1: LET X1 = X2: LET Y1 = Y2
2210 NEXT T: DRAW XB - X2, YB - Y2: LET X2 = XB: LET Y2 = YB
2220 DRAW -XB,-YB: IF H$ = "n" THEN GO TO 2250
2228 REM call hatching routine if needed.
2229 REM if angle is greater than half circle then do hatching in two parts.
2230 IF ASTO > PI THEN LET A2 = A1 - PI: LET XB = 2*PX - XA: LET YB = 2*PY -YA
     : GO SUB hatch: LET XA = XB: LET YA = YB: LET A1 = A2: LET A2 = A2S
2240 LET XB = PX + X2: LET YB = PY + Y2: GO SUB hatch
2249 REM loop back for next segment.
2250 LET A1 = A2: NEXT I: RETURN
```

requirements by using the ‘cursor’ to centre the chart and INPUTting the RADIUS in pixels. The individual data items are then INPUT and TOTalled, and this total is used to establish an angular scale for the ‘pie’-chart. Each slice is centred with the ‘cursor’, with any displacement of the ‘cursor’ from the centre of the ‘pie’ being treated as a distance along the bisector of the slice and not as an absolute position. With each new section the ‘cursor’ reappears at the original centre of the ‘pie’. Figure 6.6 was generated using this routine.

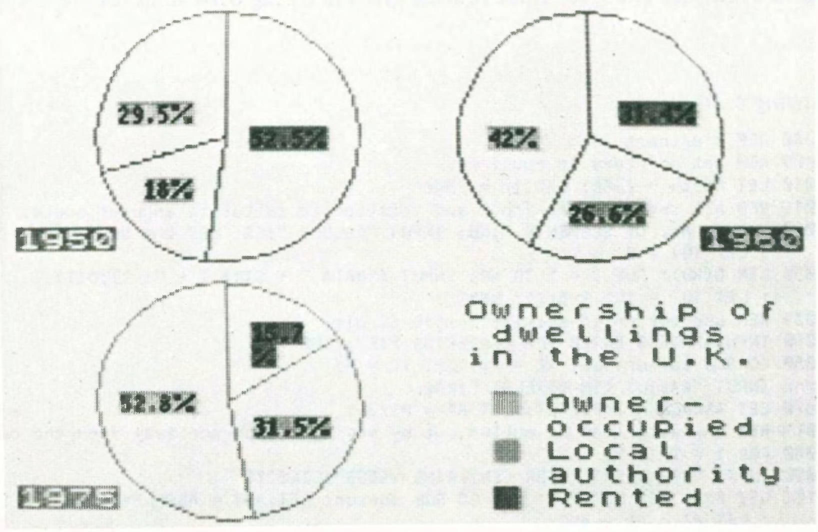


Figure 6.6

Hatching

Hatching the area of a pie-slice involves the intersection of a line with the boundaries of the slice. To make the calculations simpler we shall hatch using lines only in the horizontal direction or only in the verical direction, or both. Furthermore we hatch only ‘pie’s that subtend angles less than or equal to π radians (180 degrees) at the centre. For obtuse angles the ‘pie’ is treated as two pieces, the first subtending π radians at the centre. The ‘pie’ routine enquires whether the hatching is to be horizontal (answer “x”), vertical (answer “y”), both ways (answer “b”) or neither (answer “n”).

The pie sections we are considering are each bounded by two line segments and a circular arc. We must find which part of a hatching line (if any) lies inside this segment. Because the ‘pie’ does not subtend an angle greater than π radians at its centre there are only four possibilities

(1) a line may miss the pie altogether
(2) it may intersect the arc at two points

(3) it may intersect the arc and one of the line segments
(4) it may intersect both line segments

The special cases where the line coincidently cuts the arc and a line segment at the same point may be included in one of the above four possibilities. The explanation of the hatching algorithm is given with reference to horizontal hatching; the vertical follows in an equivalent manner. We first find the MAXimum and MINimum y-values of points within the 'pie' section. Then we consider all horizontal hatching lines with equations of the form Y = k*JUMP + REM between these limits (0 ⩽ REM ⩽ JUMP − 1). For each hatching line we calculate the two points of intersection with the extended line segments and then check whether their MU values lie between 0 and 1; that is, whether the intersection is between the centre of the circle and the arc. Next we find the two points of intersection of the hatching line with the complete circle containing the arc, and then check whether they lie on the arc. From these we can find the two points of intersection of the pie section and the hatching line, and these are then joined. This whole process is programmed in listing 6.11 and an example of its use is given in figure 6.7. Note that to fill in a slice completely we simply set JUMP equal to 1.

Listing 6.11

```
2300 REM hatch
2309 REM if cross hatching is required then run hatch routine twice.
2310 IF H$ ="b" THEN LET H$ = "x": GO SUB 2320: LET H$ = "y"
     : GO SUB 2320: LET H$ ="b" : RETURN
2319 REM set hatching variables to control direction of lines.
2320 IF H$ = "y" THEN LET PZ = PX: LET PT = PY: LET ZA = XA: LET TA = YA
     : LET ZB = XB: LET TB = YB
2330 IF H$ = "x" THEN LET PZ = PY: LET PT = PX: LET ZA = YA: LET TA = XA
     : LET ZB = YB: LET TB = XB
2340 DIM Z(3)
2349 REM find max. and min. coordinates for lines which pass through segment.
2350 LET T = PI/2: LET MAX = 0: LET MIN = 0
2360 LET VAL = SIN A1: IF H$ = "x" THEN LET VAL = COS A1
2370 IF MAX < VAL THEN LET MAX = VAL
2380 IF MIN > VAL THEN LET MIN = VAL
2390 IF T > A1 THEN LET T = T - PI/2: GO TO 2390
2400 IF T < A2 THEN GO TO 2450
2410 LET VAL = SIN T: IF H$ = "x" THEN LET VAL = COS T
2420 IF MAX < VAL THEN LET MAX = VAL
2430 IF MIN > VAL THEN LET MIN = VAL
2440 LET T = T - PI/2: GO TO 2400
2450 LET VAL = SIN A2: IF H$ = "x" THEN LET VAL = COS A2
2460 IF MAX < VAL THEN LET MAX = VAL
2470 IF MIN > VAL THEN LET MIN = VAL
2480 LET NEWMIN = INT (INT (RAD*MIN + 1)/JUMP)*JUMP + REM
2489 REM for lines which crosses segment find intersections with radii and arc.
2490 FOR E = NEWMIN TO MAX*RAD STEP JUMP
2499 REM store intersection coordinate information in array Z(1:4).
2500 LET IC = 0
2510 LET DENOM = TA - PT: IF DENOM = 0 THEN GO TO 2540
2520 LET MU = E/DENOM: IF MU < 0 OR MU > 1 THEN GO TO 2540
2530 LET IC = IC + 1: LET Z(IC) = PZ + MU*(ZA - PZ)
2540 LET DENOM = TB - PT: IF DENOM = 0 THEN GO TO 2580
2550 LET MU = E/DENOM: IF MU < 0 OR MU > 1 THEN GO TO 2580
```

```
2560 LET IC = IC + 1: LET Z(IC) = PZ + MU*(ZB - PZ)
2569 REM if more than two points of intersection found, delete duplicates.
2570 IF IC = 2 AND Z(1) = Z(2) THEN LET IC = 1
2580 IF IC <> 2 THEN GO TO 2610
2589 draw hatch lines.
2590 IF H$ = "y" THEN PLOT Z(1),E + PT: DRAW Z(2) - Z(1),0: GO TO 2710
2600 IF H$ = "x" THEN PLOT E + PT,Z(1): DRAW 0,Z(2) - Z(1): GO TO 2710
2610 LET DISC = RAD*RAD - E*E: IF DISC < 0 THEN GO TO 2710
2620 LET DISC = INT (SQR DISC + 0.5)
2630 LET ZZ = PZ + DISC: LET AZ = DISC: GO SUB in: IF OUT THEN GO TO 2650
2640 LET IC = IC + 1: LET Z(IC) = ZZ
2650 LET ZZ = PZ - DISC: LET AZ = -DISC: GO SUB in: IF OUT THEN GO TO 2670
2660 LET IC = IC + 1: LET Z(IC) = ZZ
2670 IF IC < 2 THEN GO TO 2710
2680 IF IC = 2 THEN GO TO 2590
2690 IF Z(1) = Z(2) THEN LET Z(2) = Z(3)
2700 GO TO 2590
2710 NEXT E
2720 RETURN
2800 REM in
2808 REM calculate angle from centre to point of intersection.
2809 REM if angle lies between angles of ends of segment then point
         of intersection is on the arc of the segment.
2810 LET BZ = AZ: LET EZ = E
2820 IF H$ = "x" THEN LET BZ = E: LET EZ = AZ
2830 IF BZ = 0 THEN LET PHI = -PI/2: IF EZ > 0 THEN LET PHI = -PHI
2840 IF BZ = 0 THEN GO TO 2860
2850 LET PHI = ATN (EZ/BZ): IF BZ < 0 THEN LET PHI = PHI - PI
2859 REM set flag to indicate whether or not point is valid.
2860 LET OUT = (PHI >= A1) OR (PHI <= A2)
2870 RETURN
```

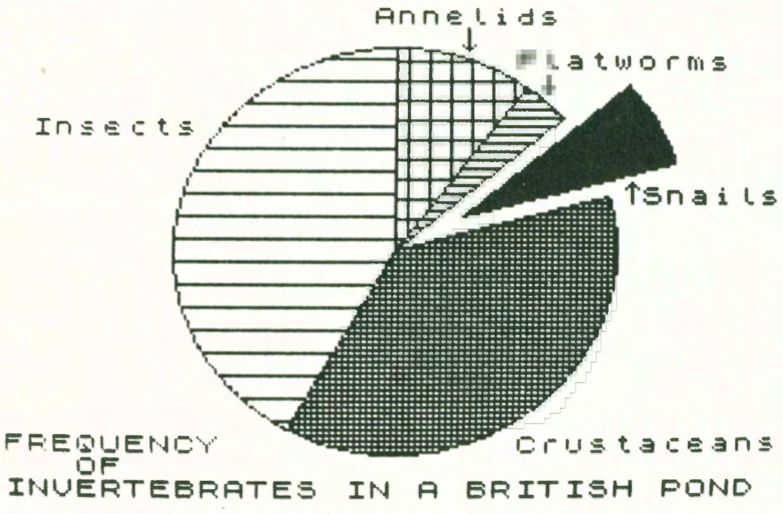


Figure 6.7

Graphs

As our final example of graphical data presentation we must consider scientific graphs of functions and graphs of discrete points. Such diagrams require coordinate axes that need not be fixed, and can cover a bewildering variety of ranges for their scales. The method we use to decide on the placing of a particular axis is fairly standard: if zero should lie in the range of the graph then the axis passes through that point, otherwise it lies on the edge of the graphics area, closest to zero. Five TICKs are then placed along each axis and the scale value at that point is written close to each TICK. The need for accuracy in scientific graphs makes us wish to include as many characters as possible. As things stand we can have only 32 characters across the screen and 22 up it, so this is where 'thin' comes in. We used the CHARACTER GENERATOR to create two sets of thin characters that are half the width of normal characters: one set having characters in the left half of the character block and the other set being in the right half. When a string is to be printed as thin characters, the 'thin' routine prints every other character of the string in the left-biased set and then OVER-prints this, with the remaining characters in the right-biased set. This places two thin characters in each block, giving us 64 print positions across the screen. The numbers to be printed as 'thin' 'label's still need to be converted into strings and made consistent in length and/or decimal accuracy. This is achieved by the routine 'number', which follows in listing 6.12. Note that the routine 'thin' has already been used in the 'label'ling of figure 6.6.

Listing 6.12

```
3000 REM graph
3009 REM symbol routine is used to mark data points on discrete graphs.
3010 LET symbol = 3500
3019 REM calculate scales and draw axes.
3020 INPUT "X GOES FROM ";XB;" TO ";XT: IF XT <= XB THEN GO TO 3020
3030 INPUT "Y GOES FROM ";YB;" TO ";YT: IF YT <= YB THEN GO TO 3030
3040 LET XSCALE = 192/(XT - XB): LET YSCALE = 128/(YT - YB)
3050 LET XO = INT (-XB*XSCALE + 32.5): LET YO = INT (-YB*YSCALE + 24.5)
3059 REM if zero is not in range of graph move axis to appropriate edge.
3060 IF YT < 0 THEN LET YO = 153
3070 IF YB > 0 THEN LET YO = 23
3080 IF XT < 0 THEN LET XO = 224
3090 IF XB > 0 THEN LET XO = 31
3100 PLOT XO,23: DRAW 0, 128: PLOT 31,YO: DRAW 192,0
3101 REM use thin routine to print labels on axes with four figure accuracy.
3110 LET XDIF = (XT - XB)/4: LET YDIF = (YT - YB)/4
3120 LET X = XB: LET Y = YB: FOR J = 1 TO 5
3130 LET PX = INT ((X - XB)*XSCALE + 32.5): LET PY = YO
3140 PLOT PX,PY-2: DRAW 0,4
3150 LET COL = INT (PX/8) - 1: LET ROW = INT ((175 - PY)/8) + 1
3160 LET A = X: GO SUB number: GO SUB thin
3170 LET PY = INT ((Y - YB)*YSCALE + 24.5): LET PX = XO
3180 PLOT PX-2,PY: DRAW 4,0
3190 LET COL = INT (PX/8) + 1: LET ROW = INT ((175 - PY)/8)
3200 LET A = Y: GO SUB number: GO SUB thin
3210 LET X = X + XDIF: LET Y = Y + YDIF: NEXT J
3220 INPUT "CONTINUOUS OR DISCRETE GRAPH ";D$: IF D$ <> "c" AND D$ <> "d"
     THEN GO TO 3220
3230 IF D$ = "d" THEN GO TO 3320
3239 REM input the function to be plotted.
3240 INPUT "F(x): y="; LINE F$
3250 LET X = XB: LET Y = VAL (F$): LET OY = INT ((Y-YB)*YSCALE + 24.5)
```

```
3260 PLOT 32,OY
3270 FOR I = 33 TO 224
3280 LET X = (I - 32)/XSCALE + XB
3290 LET Y = VAL (F$): LET IY = INT ((Y-YB)*YSCALE + 24.5)
3300 DRAW 1,IY - OY: LET OY = IY: NEXT I
3310 RETURN
3319 REM discrete graph required so input set of points.
3320 INPUT "No. OF POINTS ";NP: DIM X(NP): DIM Y(NP)
3330 FOR I = 1 TO NP: INPUT ("X(" + STR$ I + ") ");X(I);
     ("  Y(" + STR$ I + ") ");Y(I): NEXT I
3339 REM sort points into ascending order of X coordinate.
3340 FOR I = 1 TO NP - 1: FOR J = I + 1 TO NP
3350 IF X(J) < X(I) THEN LET T = X(I): LET X(I) = X(J): LET X(J) = T
     : LET T = Y(I): LET Y(I) = Y(J): LET Y(J) = T
3360 NEXT J: NEXT I
3370 LET X = INT ((X(1) - XB)*XSCALE  + 32.5)
     : LET Y = INT ((Y(1) - YB)*YSCALE + 24.5)
3380 PLOT X,Y: LET OX = X: LET OY = Y: GO SUB symbol: PLOT OX,OY
3389 REM join up points and place a 'symbol' at each point.
3390 FOR I = 2 TO NP
3400 LET X = INT ((X(I) - XB)*XSCALE  + 32.5)
     : LET Y = INT ((Y(I) - YB)*YSCALE + 24.5)
3410 PLOT X - OX,Y - OY: LET OX = X: LET OY = Y: GO SUB symbol: PLOT OX,OY
3420 NEXT I: RETURN

3500 REM symbol
3510 DRAW 0,1: DRAW 1,0: DRAW 0,-2: DRAW -2,0: DRAW 0,2
3520 RETURN

4700 REM number
4710 LET A$ = STR$ A : IF LEN A$ <= 4 THEN RETURN
4720 LET A$ = A$( TO 4)
4730 RETURN
```

Exercise 6.7

Write an extended 'number' routine that allows you to specify the format of the string to be returned. One way of doing this is to enter a string containing a template for the number format; for example, the string '##.###' could specify a number with two digits before the decimal point and three decimal places after it.

Exercise 6.8

Construct the thin 'sets' required for numerical labelling (these appear on the cassette tape as 'thin3' and 'thin4'). Use them for 'label'ling diagrams. Figure 6.8, which was drawn using listing 5.2, will help you with your construction.

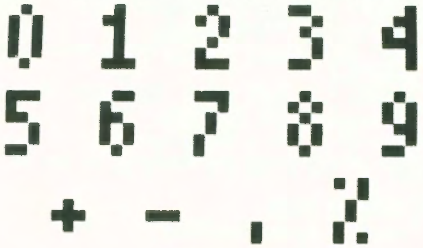


Figure 6.8

The choice is now offered between entering a functional representation of points on a continuous curve, and entering a set of discrete data points to be joined in a saw-tooth type pattern by straight lines. In the functional section of the routine the height of the line above each pixel point on the X-axis is calculated and these points are joined by lines. In the discrete section the X-coordinate and Y-coordinate of available data points are INPUT and sorted into ascending order of the X-coordinate. These points are then joined by a line. One

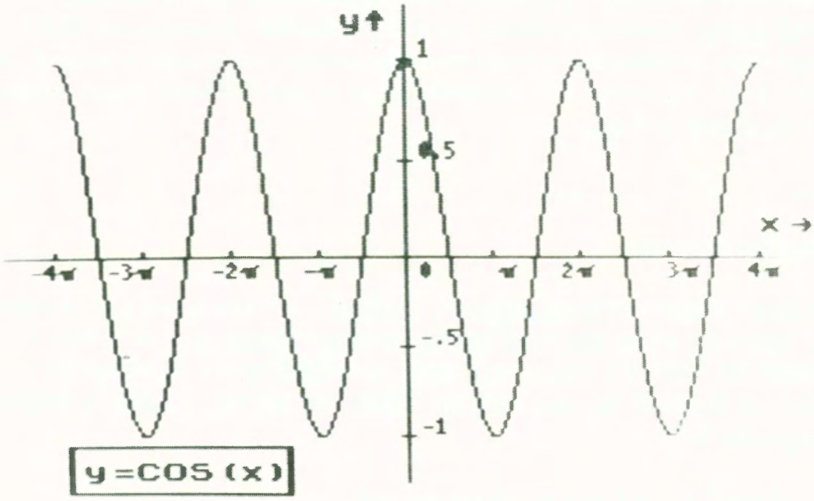


Figure 6.9

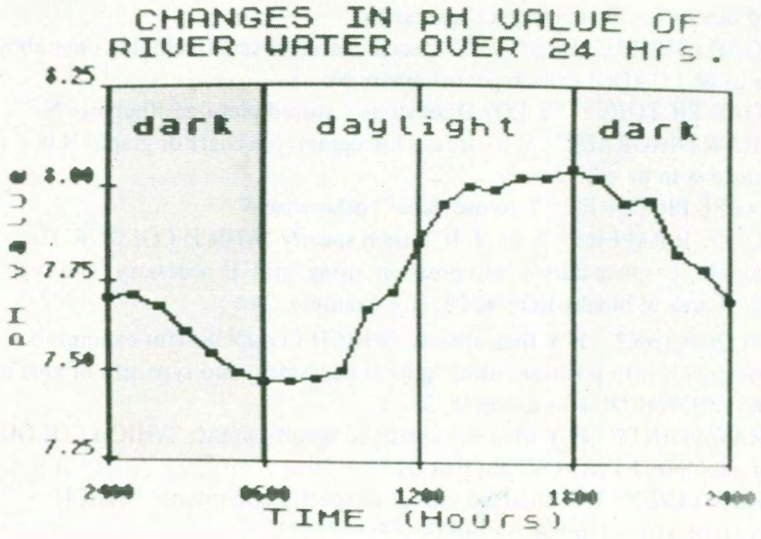


Figure 6.10

example of each type of diagram is given. Figure 6.9 shows a typical continuous cosine curve and figure 6.10 shows discrete scientific data about the pH levels of a river.

Exercise 6.9
It can be seen that the only requirement for a graph of this type is a set of co-ordinates in ascending order of X, which are then joined up. This set can be created in any manner: by a series of READ statements or by a multi-line calculation in a subroutine. Instead of eVALuating the function string F$ we could write a routine that is used every time we need to calculate a point on the curve. Produce a routine that allows the graph of SIN X/X to be drawn, avoiding the calculation SIN 0/0.

Complete Programs

We group listings 6.6 ('main program', 'charload', 'set' and 'query'), 6.1 ('cursor' and 'grid'), 6.4 ('save' and 'load') under the name 'libdiag', and use it with 6.2 ('paper' and 'ink'), 6.3 ('point' and 'line'), 6.5 ('label') and 6.7 ('create' and 'thin'): all found on the tape. Note the changes for the 16K machine mentioned in appendix A.

I. 'libdiag', 6.2, 6.3, 6.5, 6.7. Data required:

'DEFINE CHARACTERS?' (type) Yes to create special rotated characters and blocks for histograms; otherwise No
'LOAD CHARACTERS?' Y if special character sets (including ones above) are to be LOADed from tape; otherwise No
'LOAD PICTURE?' Y LOAD previously stored picture; otherwise N
'DRAW DIAGRAM?' Y to draw a histogram, pie-chart or graph; N if picture is to be edited only
'LABEL PICTURE?' Y to use 'label'; otherwise N
'COLOUR PAPER?' Y or N. If Y then specify 'WHICH COLOUR' (for example, 1: move cursor into position, using 'grid' if necessary) and type size of area in blocks ROW*COL (for example, 2*4)
'COLOUR INK?' If Y then specify 'WHICH COLOUR' (for example 5: move cursor into position, using 'grid' if necessary) and type size of area in blocks ROW*COL (for example, 3*1)
'DRAW POINT?' If Y then use cursor to specify point: 'WHICH COLOUR' (for example, 6) and OVER (1 or 0)
'DRAW LINE?' If Y then use cursor to specify end points: 'WHICH COLOUR' (for example, 6) and OVER (1 or 0)
'END PICTURE?' If N then go through sequence again
'SAVE PICTURE?' If Y then input name of picture

Experiment with the data graphs.

II. 'libdiag' and listing 6.8 ('histo'/type1). Data required: for example

'RANGE OF VERTICAL' 0 'TO' 100
'No. OF BARS' 6
'DATA FOR BAR 1' 56 'COLOUR' 2
'DATA FOR BAR 2' 95 'COLOUR' 6
'DATA FOR BAR 3' 20 'COLOUR' 4
'DATA FOR BAR 4' 77 'COLOUR' 5
'DATA FOR BAR 5' 54 'COLOUR' 1
'DATA FOR BAR 6' 33 'COLOUR' 3

III. 'libdiag' and listing 6.9 ('histo'/type2). Data required: for example

'RANGE OF VERTICAL' 0 'TO' 50
'No. OF BARS' 4
'COLOUR FOR BAR 1' 2
'DATA MAX FOR BAR 1' 44
'DATA MIN FOR BAR 1' 22
'COLOUR FOR BAR 2' 6
'DATA MAX FOR BAR 2' 36
'DATA MIN FOR BAR 2' 5
'COLOUR FOR BAR 3' 4
'DATA MAX FOR BAR 3' 42
'DATA MIN FOR BAR 3' 29
'COLOUR FOR BAR 4' 1
'DATA MAX FOR BAR 4' 31
'DATA MIN FOR BAR 4' 12

IV. 'libdiag' and listings 6.10 and 6.11 ('pie', 'hatch', etc.). Data required: for example

'No. OF SEGMENTS' 3
'COLOUR' 0
'DATA 1' 1
'DATA 2' 2
'DATA 3' 3
Centre pie with cursor then 'RADIUS (IN PIXELS)' 75
Use cursor to centre wedge 'HATCH' (x, y, b, n) b: 'JUMP' 8: 'REM' 5
Use cursor to centre wedge 'HATCH' (x, y, b, n) y: 'JUMP' 1: 'REM' 0
Use cursor to centre wedge 'HATCH' (x, y, b, n) n

V. 'libdiag' and listing 6.12 ('graph', etc.). The questions posed by the program are self-explanatory: like those above.

7 Three-dimensional Coordinate Geometry

Before we lead on to a study of the graphical display of objects in three-dimensional space, we first have to come to terms with three-dimensional Cartesian coordinate geometry. As in two-dimensional space, we arbitrarily fix a point in the space, named the *coordinate origin* (origin for short). We then imagine three mutually perpendicular lines through this point; each line goes off to infinity in both directions. These are the *x-axis, y-axis* and *z-axis*. Each axis is thought to have a positive and a negative half, both starting at the origin; that is, distances measured from the origin along the axis are positive on one side and negative on the other. We can think of the x-axis and y-axis in a similar way to two-dimensional space, both lying on the page of this book say, the positive x-axis 'horizontal' and to the right of the origin, and the positive y-axis 'vertical' and above the origin. This just leaves the position of the z-axis: it has to be perpendicular to the page (since it is perpendicular to both the x-axis and the y-axis). The positive z-axis can be into the page (the so-called *left-handed triad* of axes) or out of the page (the *right-handed triad*). *In this book we always use the left-handed triad notation*. What we say in the remainder of the book about left-handed axes has its equivalent in the right-handed system – it is not important which notation you decide finally to use, as long as you are consistent.

We specify a general point $\boldsymbol{p}$ in space by a coordinate triple or vector (X, Y, Z), where the individual coordinate values are the perpendicular projections of the point on to the respective x-axis, y-axis and z-axis. By projection we mean the unique point on the specified axis such that a line from that point to $\boldsymbol{p}$ is perpendicular to that axis.

There are two operations we need to consider for three-dimensional vectors. Suppose we have two vectors $\boldsymbol{p}_1 \equiv (x_1, y_1, z_1)$ and $\boldsymbol{p}_2 \equiv (x_2, y_2, z_2)$ then

scalar multiple $k\boldsymbol{p}_1 \equiv (k \times x_1, k \times y_1, k \times z_1)$ multiply the three individual coordinate values by a scalar number k

vector addition $\boldsymbol{p}_1 + \boldsymbol{p}_2 \equiv (x_1 + x_2, y_1 + y_2, z_1 + z_2)$ add the x-coordinates together, then the y-coordinates and finally the z-coordinates to form a new vector

$\boldsymbol{d} \equiv (x, y, z)$ and $-\boldsymbol{d} \equiv (-x, -y, -z)$ represent the same line in space but their directions are of opposite senses. We define the length of a vector $\boldsymbol{d} \equiv (x, y, z)$ (sometimes called its modulus, or absolute value) as $|\boldsymbol{d}|$, the distance of the point vector from the origin

$$|\boldsymbol{d}| = \sqrt{(x^2 + y^2 + z^2)}$$

So any point on the line $\boldsymbol{p} + \mu\boldsymbol{d}$ is found by moving to the point $\boldsymbol{p}$ and then travelling along a line that is parallel to the direction $\boldsymbol{d}$, a distance $\mu\ |\boldsymbol{d}|$ in the positive sense of $\boldsymbol{d}$ if μ is positive, and in the negative sense otherwise. Note any point on the line can act as a base vector, and the directional vector may be replaced by any non-zero scalar multiple of itself.

If the directional vector $\boldsymbol{d} \equiv (x, y, z)$ makes angles θ_x, θ_y and θ_z with the respective positive x-direction, y-direction and z-direction, then the ratios

$$x{:}y{:}z = \cos\theta_x{:}\cos\theta_y{:}\cos\theta_z$$

which means that $\boldsymbol{d} \equiv (\lambda \times \cos\theta_x, \lambda \times \cos\theta_y, \lambda \times \cos\theta_z)$ for some λ.

We know from the properties of three-dimensional geometry that

$$\cos^2\theta_x + \cos^2\theta_y + \cos^2\theta_z = 1$$

Hence $\lambda = |\boldsymbol{d}|$, and if the directional vector has unit modulus (that is, modulus $= \lambda = 1$), then the coordinates of this vector must be $(\cos\theta_x, \cos\theta_y, \cos\theta_z)$; that is, $\lambda = 1$. The coordinates of a directional vector given in this way are called the *direction cosines* of the set of lines generated by the vector. In general, if the direction vector is $\boldsymbol{d} \equiv (x, y, z)$ then the direction cosines are

$$\left(\frac{x}{|\boldsymbol{d}|}, \frac{y}{|\boldsymbol{d}|}, \frac{z}{|\boldsymbol{d}|}\right)$$

Example 7.1

Describe the line joining (1, 2, 3) to (−1, 0, 2), using the three methods shown so far.

The general point (x, y, z) on the line satisfies the equations

$$(x-1)\times(0-2) = (y-2)\times(-1-1) \quad \text{that is, } -2x + 2y = 2 \qquad (7.1)$$

$$(y-2)\times(2-3) = (z-3)\times(0-2) \qquad -y + 2z = 4 \qquad (7.2)$$

$$\text{and} \quad (z-3)\times(-1-1) = (x-1)\times(2-3) \qquad -2z + x = -5 \qquad (7.3)$$

Notice that equation 7.1 is −2 times the sum of equations 7.2 and 7.3. Thus we need consider only these latter two equations, to get

$$y = 2z - 4 \quad \text{and} \quad x = 2z - 5$$

so that the general point on the line depends only on one variable, in this case z, and is given by $(2z - 5, 2z - 4, z)$. We easily check this result by noting that when $z = 3$ we get $(1, 2, 3)$ and when $z = 2$ we get $(-1, 0, 2)$, the two original points defining the line.

In vector form the general point on the line (depending on μ) is

$$\boldsymbol{p}(\mu) \equiv (1 - \mu)(1, 2, 3) + \mu(-1, 0, 2) \equiv (1 - 2\mu, 2 - 2\mu, 3 - \mu)$$

Again the coordinates depend on just one variable (μ), and to check the validity of this representation of a line we note that $\boldsymbol{p}(0) \equiv (1, 2, 3)$ and $\boldsymbol{p}(1) \equiv (-1, 0, 2)$.

If we put the line into base/directional vector form we see that

$$\boldsymbol{p}(\mu) \equiv (1, 2, 3) + \mu(-2, -2, -1)$$

with $(1, 2, 3)$ as the base vector and $(-2, -2, -1)$ as the direction (which incidently has modulus $\sqrt{(4 + 4 + 1)} = \sqrt{9} = 3$). We noted also that any point on the line can act as a base vector, and so we can give another form for the general point on this line, $\boldsymbol{p}'$

$$\boldsymbol{p}'(\mu) \equiv (-1, 0, 2) + \mu(-2, -2, -1)$$

We can change the directional vector into its direction cosine form $(-2/3, -2/3, -1/3)$ and represent the line in another version of the base/direction form

$$\boldsymbol{p}''(\mu) \equiv (1, 2, 3) + \mu(-2/3, -2/3, -1/3)$$

Naturally the same μ value will give different points for different representations of the line; for example, $\boldsymbol{p}(3) \equiv (-5, -4, 0)$, $\boldsymbol{p}'(3) \equiv (-7, -6, -1)$ and $\boldsymbol{p}''(3) \equiv (-1, 0, 2)$. The direction of this line makes angles of 131.81 degrees $= \cos^{-1}(-2/3)$, 131.81 degrees and 109.47 degrees $= \cos^{-1}(-1/3)$ with the positive x-direction, y-direction and z-direction respectively.

The Angle Between Two Directional Vectors

In order to calculate such an angle, first we introduce the operator $\cdot$ the *dot product* or *scalar product*. This operates on two vectors and returns a scalar (real) result. Thus

$$\boldsymbol{p} \cdot \boldsymbol{q} = (x_1, y_1, z_1) \cdot (x_2, y_2, z_2) = x_1 \times x_2 + y_1 \times y_2 + z_1 \times z_2$$

If $\boldsymbol{p}$ and $\boldsymbol{q}$ are both unit vectors (that is, in direction cosine form), and θ is the

$$\boldsymbol{n} \cdot (\boldsymbol{b} + \mu \boldsymbol{d}) = k$$

that is, $\mu = (k - \boldsymbol{n} \cdot \boldsymbol{b})/(\boldsymbol{n} \cdot \boldsymbol{d})$ provided $\boldsymbol{n} \cdot \boldsymbol{d} \neq 0$.

$\boldsymbol{n} \cdot \boldsymbol{d} = 0$ if the line and plane are parallel and so either there is no point of intersection or the line is in the plane.

The Distance of a Point from a Plane

The distance of a point $\boldsymbol{p}_1$ from a plane $\boldsymbol{n} \cdot \boldsymbol{x} = k$ is the distance of $\boldsymbol{p}_1$ from the nearest point $\boldsymbol{p}_2$ on the plane. Hence the normal from the plane at $\boldsymbol{p}_2$ must pass through $\boldsymbol{p}_1$. This line can be written $\boldsymbol{p}_1 + \mu \boldsymbol{n}$, and the μ value that defines $\boldsymbol{p}_2$ is such that

$$\mu = (k - \boldsymbol{n} \cdot \boldsymbol{p}_1)/(\boldsymbol{n} \cdot \boldsymbol{n})$$

from the equation above, and the distance of the point $\boldsymbol{p}_2 \equiv \boldsymbol{p}_1 + \mu \boldsymbol{n}$ from $\boldsymbol{p}_1$ is

$$\mu \times |\boldsymbol{n}| = |k - \boldsymbol{n} \cdot \boldsymbol{p}_1|/|\boldsymbol{n}|$$

In particular, if $\boldsymbol{p}_1$ is the origin $\boldsymbol{O}$ then the distance of the plane from the origin is $|k|/|\boldsymbol{n}|$. Furthermore, if $\boldsymbol{n}$ is a direction cosine vector, we see that the distance of the origin from the plane is $|k|$, the absolute value of the real number k.

Example 7.2

Find the point of intersection of the line joining $(1, 2, 3)$ to $(-1, 0, 2)$ with the plane $(0, -2, 1) \cdot \boldsymbol{x} = 5$, and also find the distance of the plane from the origin.

$$\begin{aligned}
\boldsymbol{b} &\equiv (1, 2, 3) \\
\boldsymbol{n} &\equiv (0, -2, 1) \\
\boldsymbol{d} &\equiv (-1, 0, 2) - (1, 2, 3) \equiv (-2, -2, -1) \\
\boldsymbol{n} \cdot \boldsymbol{b} &= (0 \times 1 + -2 \times 2 + 1 \times 3) = -1 \\
\boldsymbol{n} \cdot \boldsymbol{d} &= (0 \times -2 + -2 \times -2 + 1 \times -1) = 3
\end{aligned}$$

hence the μ value of the point of intersection is $(5 - (-1))/3 = 2$, and the point vector is

$$(1, 2, 3) + 2(-2, -2, -1) \equiv (-3, -2, 1)$$

and the distance from the origin is $5/|\boldsymbol{n}| = 5/\sqrt{5} = \sqrt{5}$.

The program given in listing 7.1 enables us to calculate the point of intersection (array P) of a line and a plane. The line has base vector B and direction D, and the plane has normal N and plane constant K. Note, since we are working with decimal numbers, and thus are subject to rounding errors, we cannot check if a dot product is zero. We can find only if it is sufficiently small to be considered zero, and what is meant by sufficiently small is left to the programmer (on the Spectrum about six places after the decimal point is reasonable).

Listing 7.1

```
100 REM intersection of line and plane
110 DIM B(3): DIM D(3): DIM N(3): DIM P(3): DIM A$(8)
120 INPUT "BASE VECTOR OF LINE","(";B(1);",";B(2);",";B(3);")"
130 PRINT AT 1,0;"BASE VECTOR OF LINE","(";B(1);",";B(2);",";B(3);")"
140 INPUT "DIRECTION VECTOR OF LINE","(";D(1);",";D(2);",";D(3);")"
150 PRINT AT 4,0;"DIRECTION VECTOR OF LINE","(";D(1);",";D(2);",";D(3);")"
160 INPUT "NORMAL TO PLANE",,"(";N(1);",";N(2);",";N(3);")"
170 PRINT AT 7,0;"NORMAL TO PLANE",,"(";N(1);",";N(2);",";N(3);")"
180 INPUT "PLANE CONSTANT ";K
188 REM calculate point of intersection (P(1),P(2),P(3))
189 REM of line and plane, data input above.
190 PRINT AT 10,0;"PLANE CONSTANT ";K
200 LET DOT = N(1)*D(1) + N(2)*D(2) + N(3)*D(3)
209 REM zero dot product so no intersection.
210 IF ABS DOT < 0.000001 THEN PRINT AT 15,0;"NO POINT OF INTERSECTION": STOP
220 LET MU = (K - N(1)*B(1) - N(2)*B(2) - N(3)*B(3))/DOT
230 FOR I = 1 TO 3: LET P(I) = B(I) + MU*D(I)
    : IF ABS P(I) < 0.000001 THEN LET P(I) =  0
240 NEXT I: PRINT AT 15,0;"POINT OF INTERSECTION ","(";
249 REM tidy up output.
250 FOR I = 1 TO 3: LET A$ = STR$ P(I): FOR J = 1 TO 8
260 IF A$(J) <> " " THEN PRINT A$(J);
270 NEXT J: IF I <> 3 THEN PRINT ",";
280 NEXT I: PRINT ")"
290 STOP
```

The Point of Intersection of Two Lines

Suppose we have two lines $\boldsymbol{b}_1 + \mu\boldsymbol{d}_1$ and $\boldsymbol{b}_2 + \lambda\boldsymbol{d}_2$. Their point of intersection, if it exists (if the lines are not coplanar or are parallel then they will not intersect), is identified by finding unique values for μ and λ that satisfy the vector equation (three separate coordinate equations)

$$\boldsymbol{b}_1 + \mu\boldsymbol{d}_1 = \boldsymbol{b}_2 + \lambda\boldsymbol{d}_2$$

Three equations in two unknowns means that for the equations to be meaningful there must be at least one pair of equations that are independent, and the remaining equation must be a combination of these two. Two lines are parallel if one directional vector is a scalar multiple of the other. So we take two independent equations, find the values of μ and λ (we have two equations in two unknowns), and put them in the third equation to see if they are consistent. Example 7.3, below, demonstrates this method, and listing 7.2 is a way of implementing it on a computer. The first line has base and direction stored in arrays B and D, and the second line in C and E: the calculated point of intersection goes into array P.

Note that if the two independent equations are

$$a_{11} \times \mu + a_{12} \times \lambda = b_1$$

$$a_{21} \times \mu + a_{22} \times \lambda = b_2$$

then the *determinant* of this pair of equations $\Delta = a_{11} \times a_{22} - a_{12} \times a_{21}$, will be non-zero (because the equations are not related), and we have the solutions

$$\mu = (a_{22} \times b_1 - a_{12} \times b_2)/\Delta \quad \text{and} \quad \lambda = (a_{11} \times b_2 - a_{21} \times b_1)/\Delta$$

Listing 7.2

```
100 REM intersection of two lines
110 DIM B(3): DIM D(3): DIM C(3): DIM E(3): DIM P(3): DIM A$(8)
120 INPUT "BASE VECTOR OF FIRST LINE","(";B(1);",";B(2);",";B(3);")"
130 PRINT AT 1,0;"BASE VECTOR OF FIRST LINE","(";B(1);",";B(2);",";B(3);")"
140 INPUT "DIRECTION VECTOR OF FIRST LINE","(";D(1);",";D(2);",";D(3);")"
150 PRINT AT 4,0;"DIRECTION VECTOR OF FIRST LINE","(";D(1);",";D(2);",";D(3);")"
160 INPUT "BASE VECTOR OF SECOND LINE","(";C(1);",";C(2);",";C(3);")"
170 PRINT AT 7,0;"BASE VECTOR OF SECOND LINE","(";C(1);",";C(2);",";C(3);")"
180 INPUT "DIRECTION VECTOR OF SECOND LINE","(";E(1);",";E(2);",";E(3);")"
190 PRINT AT 10,0;"DIRECTION VECTOR OF SECOND LINE","(";E(1);",";E(2);",";E(3);")"
198 REM calculate point of intersection (P(1),P(2),P(3))
199 REM of two lines, data input above.
195 REM find any two independent line equations from the three (x/y/z).
200 FOR I = 1 TO 3
210 LET J = I + 1: IF J = 4 THEN LET J = 1
220 LET DELTA = E(I)*D(J) - E(J)*D(I)
230 IF ABS DELTA > 0.000001 THEN GO TO 260
240 NEXT I
249 REM cannot find two independent equations: lines do not intersect.
250 PRINT AT 15,0;"LINES DO NOT INTERSECT": STOP
259 REM calculate MU and LAMBDA values of point of intersection.
260 LET MU = (E(I)*(C(J) - B(J)) - E(J)*(C(I) - B(I)))/DELTA
270 LET LAMBDA = (D(I)*(C(J) - B(J)) - D(J)*(C(I) - B(I)))/DELTA
279 REM no solution if MU and LAMBDA do not satisfy third equation.
280 LET K = J + 1: IF K = 4 THEN LET K = 1
290 IF ABS (B(K) + MU*D(K) - C(K) - LAMBDA*E(K)) > 0.000001 THEN GO TO 250
299 REM calculate (P(1),P(2),P(3)) using MU value.
300 FOR I = 1 TO 3: LET P(I) = B(I) + MU*D(I)
    : IF ABS P(I) < 0.000001 THEN LET P(I) = 0
310 NEXT I: PRINT AT 15,0;"POINT OF INTERSECTION ","(";
319 REM tidy up output.
320 FOR I = 1 TO 3: LET A$ = STR$ P(I): FOR J = 1 TO 8
330 IF A$(J) <> " " THEN PRINT A$(J);
340 NEXT J: IF I <> 3 THEN PRINT ",";
350 NEXT I: PRINT ")"
360 STOP
```

Example 7.3
Find the point of intersection (if any) of

(a) $(1, 1, 1) + \mu(2, 1, 3)$ with $(0, 0, 1) + \lambda(-1, 1, 1)$,
(b) $(2, 3, 4) + \mu(1, 1, 1)$ with $(-2, -3, -4) + \lambda(1, 2, 3)$.

In (a) the three equations are

$$1 + 2\mu = 0 - \lambda \tag{7.4}$$

$$1 + \mu = 0 + \lambda \tag{7.5}$$

$$1 + 3\mu = 1 + \lambda \tag{7.6}$$

From equations 7.4 and 7.5 we get $\mu = -2/3$ and $\lambda = 1/3$, which when substituted in equation 7.6 gives $1 + 3 \times (-2/3) = -1$ on the left-hand side and $1 + 1 \times (1/3) = 4/3$ on the right-hand side, which are obviously unequal, so the lines do not intersect. From (b) we get the equations

$$2 + \mu = -2 + \lambda \tag{7.7}$$

$$3 + \mu = -3 + 2\lambda \tag{7.8}$$

$$4 + \mu = -4 + 3\lambda \tag{7.9}$$

and from equations 7.7 and 7.8 we get $\mu = -2$ and $\lambda = 2$, and these values also satisfy equation 7.9 (left-hand side = right-hand side = 2). So the point of intersection is

$$(2, 3, 4) + -2(1, 1, 1) = (-2, -3, -4) + 2(1, 2, 3) = (0, 1, 2)$$

The Plane Through Three Non-collinear Points

In order to solve this problem we must introduct a new vector operator, $\times$ the vector product, which operates on two vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ (say) giving the vector result

$$\begin{aligned}\boldsymbol{p} \times \boldsymbol{q} &= (p_1, p_2, p_3) \times (q_1, q_2, q_3)\\ &= (p_2 \times q_3 - p_3 \times q_2, p_3 \times q_1 - p_1 \times q_3, p_1 \times q_2 - p_2 \times q_1)\end{aligned}$$

If $\boldsymbol{p}$ and $\boldsymbol{q}$ are non-parallel directional vectors then $\boldsymbol{p} \times \boldsymbol{q}$ is the directional vector perpendicular to both $\boldsymbol{p}$ and $\boldsymbol{q}$. It should be noted also that this operation is *non-commutative*. That is, in general for given values of $\boldsymbol{p}$ and $\boldsymbol{q}$, we note that $\boldsymbol{p} \times \boldsymbol{q} \neq \boldsymbol{q} \times \boldsymbol{p}$. These two vector products will represent directions on the same line but with opposite senses. For example, $(1, 0, 0) \times (0, 1, 0) = (0, 0, 1)$ but $(0, 1, 0) \times (1, 0, 0) = (0, 0, -1)$; $(0, 0, 1)$ and $(0, 0, -1)$ are both parallel to the z-axis (and so perpendicular to the directions $(1, 0, 0)$ and $(0, 1, 0)$), but they are of opposite senses. Listing 7.3 gives a main program that calls the routines 'vecprod' (for the vector product of two vectors L and M returning vector N) and 'dotprod' (which calculates the dot product DOT of the vectors L and M), both given in listing 7.4.

Listing 7.3

```
100 REM dot product and vector product
110 LET vecprod = 300: LET dotprod = 400
120 DIM L(3): DIM M(3): DIM N(3)
130 INPUT "VECTOR L","(";L(1);",";L(2);",";L(3);")"
140 PRINT AT 1,0;"VECTOR L","(";L(1);",";L(2);",";L(3);")"
150 INPUT "VECTOR M","(";M(1);",";M(2);",";M(3);")"
160 PRINT AT 4,0;"VECTOR M","(";M(1);",";M(2);",";M(3);")"
170 GO SUB vecprod
180 PRINT AT 8,0;"VECTOR PRODUCT","(";N(1);",";N(2);",";N(3);")"
190 GO SUB dotprod
200 PRINT AT 11,0;"DOT PRODUCT",DOT
210 STOP
```

Listing 7.4

```
300 REM vecprod
301 REM IN  : L(3),M(3)
302 REM OUT : N(3)
309 REM N is the vector product of L and M.
310 LET NI = 2: LET NNI = 3
320 FOR I = 1 TO 3
330 LET N(I) = L(NI)*M(NNI)-L(NNI)*M(NI)
340 LET NI = NNI: LET NNI = NI + 1: IF NNI = 4 THEN LET NNI = 1
350 NEXT I
360 RETURN

400 REM dotprod
401 REM IN  : L(3),M(3)
402 REM OUT : DOT
409 REM DOT is the dot product of L and M.
410 LET DOT = 0
420 FOR I = 1 TO 3: LET DOT = DOT + L(I)*M(I): NEXT I
430 RETURN
```

Suppose we are given three non-collinear points $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$. Then the two vectors $\boldsymbol{p}_2 - \boldsymbol{p}_1$ and $\boldsymbol{p}_3 - \boldsymbol{p}_1$ represent the directions of two lines coincident at $\boldsymbol{p}_1$, both of which lie in the plane containing the three points. We know that the normal to the plane is perpendicular to every line in the plane, in particular the two lines mentioned above. Also, because the points are not collinear $\boldsymbol{p}_2 - \boldsymbol{p}_1 \neq \boldsymbol{p}_3 - \boldsymbol{p}_1$, the normal to the plane is $(\boldsymbol{p}_2 - \boldsymbol{p}_1) \times (\boldsymbol{p}_3 - \boldsymbol{p}_1)$, and since $\boldsymbol{p}_1$ lies in the plane the equation is

$$((\boldsymbol{p}_2 - \boldsymbol{p}_1) \times (\boldsymbol{p}_3 - \boldsymbol{p}_1)) \cdot (\boldsymbol{x} - \boldsymbol{p}_1) = 0$$

Example 7.4
Give the coordinate equation of the plane through the points (0, 1, 1), (1, 2, 3) and (−2, 3, −1).

This is given by the general point $\boldsymbol{x} \equiv (x, y, z)$ where

$$(((1, 2, 3) - (0, 1, 1)) \times ((-2, 3, -1) - (0, 1, 1))) \cdot ((x, y, z) - (0, 1, 1)) = 0$$

that is

$$((1, 1, 2) \times (-2, 2, -2) \cdot (x, y - 1, z - 1) = 0$$

or

$$(-6, -2, 4) \cdot (x, y - 1, z - 1) = 0$$

which in coordinate form is $-6x - 2y + 4z - 2 = 0$ or in the equivalent form $3x + y - 2z = -1$.

The Point of Intersection of Three Planes

We assume that the three planes are defined by equations 7.10 to 7.12 below. The point of intersection of these three planes, $\boldsymbol{x} \equiv (x, y, z)$, must lie in all three planes and satisfy

$$\boldsymbol{n}_1 \cdot \boldsymbol{x} = k_1 \qquad (7.10)$$

$$\boldsymbol{n}_2 \cdot \boldsymbol{x} = k_2 \qquad (7.11)$$

$$\boldsymbol{n}_3 \cdot \boldsymbol{x} = k_3 \qquad (7.12)$$

where $\boldsymbol{n}_1 \equiv (n_{11}, n_{12}, n_{13}), \boldsymbol{n}_2 \equiv (n_{21}, n_{22}, n_{23})$ and $\boldsymbol{n}_3 \equiv (n_{31}, n_{32}, n_{33})$. We can rewrite these three equations as one matrix equation

$$\begin{pmatrix} n_{11} & n_{12} & n_{13} \\ n_{21} & n_{22} & n_{23} \\ n_{31} & n_{32} & n_{33} \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \end{pmatrix}$$

and so the solution for $\boldsymbol{x}$ is given by the *column vector*

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} n_{11} & n_{12} & n_{13} \\ n_{21} & n_{22} & n_{23} \\ n_{31} & n_{32} & n_{33} \end{pmatrix}^{-1} \times \begin{pmatrix} k_1 \\ k_2 \\ k_3 \end{pmatrix}$$

So any calculation requiring the intersection of three planes necessarily involves the inversion of a 3 × 3 matrix. Listing 7.5 gives the Adjoint method of finding M, the inverse of matrix N.

Listing 7.5

```
500 REM inv/erse of 3x3 matrix
501 IN  : N(3,3)
502 OUT : SINGULAR, M(3,3)
510 LET SINGULAR = 1
520 LET DET = 0: LET NI = 2: LET NNI = 3
530 FOR I = 1 TO 3
540 LET DET = DET + N(1,I)*(N(2,NI)*N(3,NNI)-N(2,NNI)*N(3,NI))
550 LET NI = NNI: LET NNI = NI + 1: IF NNI = 4 THEN LET NNI = 1
560 NEXT I
569 REM DETerminant of singular matrix is zero, there is no inverse.
570 IF ABS DET < 0.000001 THEN RETURN
579 REM calculate M, the inverse of N, by the Adjoint method.
580 LET NI = 2: LET NNI = 3
590 FOR I = 1 TO 3
600 LET NJ = 2: LET NNJ = 3
610 FOR J = 1 TO 3
620 LET M(J,I) = (N(NI,NJ)*N(NNI,NNJ) - N(NI,NNJ)*N(NNI,NJ))/DET
630 LET NJ = NNJ: LET NNJ = NJ + 1: IF NNJ = 4 THEN LET NNJ = 1
640 NEXT J
650 LET NI = NNI: LET NNI = NI + 1: IF NNI = 4 THEN LET NNI = 1
660 NEXT I
670 LET SINGULAR = 0
680 RETURN
```

Again in this routine, vectors are represented as one-dimensional arrays, thus B(3) contains the solution of the equations, $\boldsymbol{x}$, while K(3) contains the plane constants. We are given the normals $\boldsymbol{n}_1$, $\boldsymbol{n}_2$ and $\boldsymbol{n}_3$ in the form of a 3 X 3 array N, so the values in B are found by the following code. Obviously if any two of

Listing 7.6

```
100 REM intersection of three planes
110 DIM N(3,3): DIM M(3,3): DIM K(3): DIM B(3)
120 LET inv = 500
129 REM input data on three planes.
130 PRINT AT 2,10;"COEFFICIENTS  CONSTANT"
140 FOR I = 1 TO 3
150 PRINT AT 2 + 2*I,0;"PLANE(";I;") = (   ,   ,   )"
160 FOR J = 1 TO 3
170 LET I$ = "INPUT  N(" + STR$ I + "," + STR$ J + ") "
180 INPUT (I$);N(I,J): PRINT AT 2 + 2*I,7 + 4*J;N(I,J)
190 NEXT J
200 LET I$ = "INPUT  K(" + STR$ I + ") "
210 INPUT (I$);K(I): PRINT AT 2 + 2*I,28;K(I)
220 NEXT I
229 REM if matrix of normals is singular then no intersection.
230 GO SUB inv
240 PRINT AT 12,3;"POINT OF INTERSECTION"
250 IF SINGULAR THEN PRINT AT 12,0;"NO": STOP
259 REM point of intersection is (B(1),B(2),B(3)).
260 FOR I = 1 TO 3
270 LET B(I) = 0
280 FOR J = 1 TO 3
290 LET B(I) = B(I) + M(I,J)*K(J)
300 NEXT J
310 IF ABS B(I) < 0.000001 THEN LET B(I) = 0
320 PRINT AT 12 + 2*I,7;"B(";I;") = ";B(I)
330 NEXT I
340 STOP
```

the planes are parallel or the three meet in a line, then there is no unique point of intersection: in these cases DET, the *determinant* of the matrix N is zero and variable SINGULAR = 1. (See listing 7.6.)

Example 7.5

Find the point of intersection of the three planes $(0, 1, 1) \cdot \boldsymbol{x} = 2$, $(1, 2, 3) \cdot \boldsymbol{x} = 4$ and $(1, 1, 1) \cdot \boldsymbol{x} = 0$.

In the matrix form we have

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 0 \end{pmatrix}$$

The inverse of $\begin{pmatrix} 0 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}$ is $\begin{pmatrix} -1 & 0 & 1 \\ 2 & -1 & 1 \\ -1 & 1 & -1 \end{pmatrix}$

and so

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -1 & 0 & 1 \\ 2 & -1 & 1 \\ -1 & 1 & -1 \end{pmatrix} \times \begin{pmatrix} 2 \\ 4 \\ 0 \end{pmatrix} = \begin{pmatrix} -2 \\ 0 \\ 2 \end{pmatrix}$$

This solution is easily checked: $(0, 1, 1) \cdot (-2, 0, 2) = 2$, $(1, 2, 3) \cdot (-2, 0, 2) = 4$ and $(1, 1, 1) \cdot (-2, 0, 2) = 0$, which means the point $(-2, 0, 2)$ lies on all three planes and so is their point of intersection.

The Line of Intersection of Two Planes

Let the two planes be

$$\boldsymbol{p} \cdot \boldsymbol{x} = (p_1, p_2, p_3) \cdot \boldsymbol{x} = k_1 \text{ and}$$
$$\boldsymbol{q} \cdot \boldsymbol{x} = (q_1, q_2, q_3) \cdot \boldsymbol{x} = k_2$$

We assume that the planes are not parallel, and so $\boldsymbol{p} \neq \lambda \boldsymbol{q}$ for all λ. The line common to the two planes naturally lies in each plane, and so it must be perpendicular to the normals of both planes ($\boldsymbol{p}$ and $\boldsymbol{q}$). Thus the direction of this line must be $\boldsymbol{d} \equiv \boldsymbol{p} \times \boldsymbol{q}$ and the line can be written in the form $\boldsymbol{b} + \mu \boldsymbol{d}$, where $\boldsymbol{b}$ can be any point on the line. In order to classify completely the line, we have to find one such $\boldsymbol{b}$. We find a point that is the intersection of the two planes, together with a third that is neither parallel to them nor cuts them in a common line. Choosing a plane with normal $\boldsymbol{p} \times \boldsymbol{q}$ will satisfy these conditions (and remember that we have already calculated this vector product). We still need a value for k_3, but any will do, so we take $k_3 = 0$ in order that this third plane goes through the origin. Thus $\boldsymbol{b}$ is given by the column vector

$$\boldsymbol{b} = \begin{pmatrix} p_1 & p_2 & p_3 \\ q_1 & q_2 & q_3 \\ p_2 \times q_3 - p_3 \times q_2 & p_3 \times q_1 - p_1 \times q_3 & p_1 \times q_2 - p_2 \times q_1 \end{pmatrix}^{-1} \times \begin{pmatrix} k_1 \\ k_2 \\ 0 \end{pmatrix}$$

Example 7.6

Find the line common to the planes $(0, 1, 1) \cdot \boldsymbol{x} = 2$ and $(1, 2, 3) \cdot \boldsymbol{x} = 2$. Since $\boldsymbol{p} = (0, 1, 1)$ and $\boldsymbol{q} = (1, 2, 3)$, then $\boldsymbol{p} \times \boldsymbol{q} = (1 \times 3 - 1 \times 2, 1 \times 1 - 0 \times 3, 0 \times 2 - 1 \times 1) = (1, 1, -1)$. We require the inverse of

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 1 & -1 \end{pmatrix} = \frac{1}{3}\begin{pmatrix} -5 & 2 & 1 \\ 4 & -1 & 1 \\ -1 & 1 & -1 \end{pmatrix}$$

and hence the point of intersection of the three planes is

$$\frac{1}{3}\begin{pmatrix} -5 & 2 & 1 \\ 4 & -1 & 1 \\ -1 & 1 & -1 \end{pmatrix} \times \begin{pmatrix} 2 \\ 2 \\ 0 \end{pmatrix} = \frac{1}{3}\begin{pmatrix} -6 \\ 6 \\ 0 \end{pmatrix} = \begin{pmatrix} -2 \\ 2 \\ 0 \end{pmatrix}$$

and the line is $(-2, 2, 0) + \mu(1, 1, -1)$.

It is easy to check this result, because all the points on the line should lie in both planes

$$(0, 1, 1) \cdot ((-2, 2, 0) + \mu(1, 1, -1)) = (0, 1, 1) \cdot (-2, 2, 0) + \mu(0, 1, 1) \cdot$$
$$\cdot (1, 1, -1) = 2 \qquad \text{for all } \mu \text{ and}$$
$$(1, 2, 3) \cdot ((-2, 2, 0) + \mu(1, 1, -1)) = (0, 1, 1) \cdot (-2, 2, 0) + \mu(1, 2, 3) \cdot$$
$$\cdot (1, 1, -1) = 2 \qquad \text{for all } \mu$$

The program to solve this problem (listing 7.7) is given below; note it is very similar to the previous program. Also note that arrays are not explicitly used for $\boldsymbol{p}$ and $\boldsymbol{q}$, these values are stored in the first two rows of array N. Array B holds the base vector of the line of intersection, but we do not place $\boldsymbol{d}$ in an array because the values are already in the third row of N.

Functional Representation of a Surface

In our study of two-dimensional space in chapter 3 we noted that curves can be represented in a functional notation. This idea can be extended into three

Listing 7.7

```
100 REM Line of intersection of two planes
110 DIM N(3,3): DIM M(3,3): DIM K(3): DIM B(3): DIM A$(2)
120 LET inv = 500: LET A$ = "PQ"
129 REM input data on two planes.
130 PRINT AT 2,10;"COEFFICIENTS  CONSTANT"
140 FOR I = 1 TO 2
150 PRINT AT 2 + 2*I,0;"PLANE(";I;")= (   ,   ,   )"
160 FOR J = 1 TO 3
170 LET I$ = "INPUT  " + A$(I) + "(" + STR$ J + ") "
180 INPUT (I$);N(I,J): PRINT AT 2 + 2*I,7 + 4*J;N(I,J)
190 NEXT J
200 LET I$ = "INPUT  K(" + STR$ I + ") "
210 INPUT (I$);K(I): PRINT AT 2 + 2*I,28;K(I)
220 NEXT I
229 REM form third plane.
230 LET N(3,1) = N(1,2)*N(2,3) - N(1,3)*N(2,2)
240 LET N(3,2) = N(1,3)*N(2,1) - N(1,1)*N(2,3)
250 LET N(3,3) = N(1,1)*N(2,2) - N(1,2)*N(2,1)
260 LET K(3) = 0
269 REM if matrix of normals is singular then no intersection.
270 GO SUB inv
280 PRINT AT 10,3;"LINE OF INTERSECTION"
290 IF SINGULAR THEN PRINT AT 10,0;"NO": STOP
300 PRINT AT 12,2;"BASE VECTOR       DIRECTION"
308 REM Line of intersection :-
309 REM base (B(1),B(2),B(3)) and direction (N(3,1),N(3,2),N(3,3)).
310 FOR I = 1 TO 3
320 LET B(I) = 0
330 FOR J = 1 TO 3
340 LET B(I) = B(I) + M(I,J)*K(J)
350 NEXT J
360 IF ABS B(I) < 0.000001 THEN LET B(I) = 0
370 PRINT AT 12 + 2*I,0;"B(";I;") = ";B(I)
380 PRINT AT 12 + 2*I,20;"D(";I;") = ";N(3,I)
390 NEXT I
400 STOP
```

dimensions when we study surfaces. The simplest form of surface is an infinite plane with normal $\boldsymbol{n} \equiv (n_1, n_2, n_3)$, which we have seen can be given as a coordinate equation

$$\boldsymbol{n} \cdot \boldsymbol{x} - k = n_1 \times x + n_2 \times y + n_3 \times z - k = 0$$

This can be rewritten in functional form for a general point $\boldsymbol{x} \equiv (x, y, z)$ on the curve

$$f(\boldsymbol{x}) \equiv f(x, y, z) \equiv n_1 \times x + n_2 \times y + n_3 \times z - k \equiv \boldsymbol{n} \cdot \boldsymbol{x} - k$$

This is a simple expression in variables x, y and z ($\boldsymbol{x}$) that enables us to divide all the points in space into three sets, those with $f(\boldsymbol{x}) = 0$ (the zero set), with $f(\boldsymbol{x}) < 0$ (the negative set) and $f(\boldsymbol{x}) > 0$ (the positive set). A point $\boldsymbol{x}$ lies on the surface if and only if it belongs to the zero set. If the surface divides space into two halves (each half being *connected*; that is, any two points in a given half can be joined by a curve that does not cross the surface), then these two halves may be identified with the positive and negative sets. Again beware, there are many curves that divide space into more than two connected areas and then it is impossible to relate functional representation with connected sets; for example, $f(x, y, z) = \cos(y) - \sin(x^2 + z^2)$. There are, however, many useful well-behaved curves with this property, the sphere of radius r for example

$$f(\boldsymbol{x}) = r^2 - |\boldsymbol{x}|^2$$

that is,

$$f(x, y, z) = r^2 - x^2 - y^2 - z^2$$

If $f(\boldsymbol{x}) = 0$ then $\boldsymbol{x}$ lies on the sphere, if $f(\boldsymbol{x}) < 0$ then $\boldsymbol{x}$ lies outside the sphere, and if $f(\boldsymbol{x}) > 0$ then $\boldsymbol{x}$ lies inside the sphere.

The functional representation of a surface is a very useful concept. It can be used to define sets of equations necessary in calculating the intersections of various objects. The major use, however, is to determine whether or not two points $\boldsymbol{p}$ and $\boldsymbol{q}$ (say) lie on the same side of a surface that divides space in two. All we need do is compare the signs of $f(\boldsymbol{p})$ and $f(\boldsymbol{q})$. If they are of opposite signs then a line joining $\boldsymbol{p}$ and $\boldsymbol{q}$ must cut the surface. An example is given below.

Is a Point on the Same Side of a Plane as the Origin?

Suppose the plane is defined (as earlier) by three non-collinear points $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$. Then the equation of the plane is

$$((\boldsymbol{p}_2 - \boldsymbol{p}_1) \times (\boldsymbol{p}_3 - \boldsymbol{p}_1)) \cdot (\boldsymbol{x} - \boldsymbol{p}_1) = 0$$

We may rewrite this in functional form

$$f(\boldsymbol{x}) \equiv ((\boldsymbol{p}_2 - \boldsymbol{p}_1) \times (\boldsymbol{p}_3 - \boldsymbol{p}_1)) \cdot (\boldsymbol{x} - \boldsymbol{p}_1)$$

So all we need do for a point $\boldsymbol{e}$ (say) is to compare $f(\boldsymbol{e})$ with $f(\boldsymbol{O})$, where $\boldsymbol{O}$ is the origin. We assume here that neither $\boldsymbol{O}$ nor $\boldsymbol{e}$ lie in the plane.

We shall see that this idea is of great use in the study of hidden line algorithms.

Example 7.7
Are the origin and point (1, 1, 3) on the same side of the plane defined by points (0, 1, 1), (1, 2, 3) and (−2, 3, −1)?

From example 7.4 we see that the functional representation of the plane is

$$f(\boldsymbol{x}) \equiv ((-6, -2, 4) \cdot (\boldsymbol{x} - (0, 1, 1))$$

Thus

$$f(0, 0, 0) = -(-6, -2, 4) \cdot (0, 1, 1) = -2$$

and

$$f(1, 1, 3) = -(-6, -2, 4) \cdot ((1, 1, 3) - (0, 1, 1)) = 2$$

Hence $(1, 1, 3)$ lies on the opposite side of the plane to the origin and so a line segment joining the two points will cut the plane at a point $(1 - \mu)(0, 0, 0) + \mu(1, 1, 3)$ where $0 < \mu < 1$.

Is an Oriented Convex Polygon of Vertices in Two-dimensional Space Clockwise or Anti-clockwise?

We start by assuming that the polygon is a triangle defined by the three vertices $\boldsymbol{p}_1 \equiv (x_1, y_1), \boldsymbol{p}_2 \equiv (x_2, y_2)$ and $\boldsymbol{p}_3 \equiv (x_3, y_3)$. Although these points are in two-dimensional space we can assume they lie in the x/y plane through the origin of three-dimensional space by giving them all a z-coordinate value of zero. We systematically define the directions of the edges of the polygon to be $(\boldsymbol{p}_2 - \boldsymbol{p}_1), (\boldsymbol{p}_3 - \boldsymbol{p}_2)$ and $(\boldsymbol{p}_1 - \boldsymbol{p}_3)$. Since these lines all lie in the x/y plane through the origin we know that for all $i = 1, 2$ or 3 and for some real numbers r_i that depend on i

$$(\boldsymbol{p}_{i+1} - \boldsymbol{p}_i) \times (\boldsymbol{p}_{i+2} - \boldsymbol{p}_{i+1}) = (0, 0, r_i)$$

This is because this vector product is perpendicular to the x/y plane and so only z-coordinate values can be non-zero. The addition of subscripts is modulo 3. Because the vertices were taken systematically, note that the signs of these r_i values are always the same – but what is more important, if the $\boldsymbol{p}_i$ are clockwise then the r_i are all negative, and if the $\boldsymbol{p}_i$ are anti-clockwise the r_i are all positive.

Given an oriented convex polygon, we need consider only the first three vertices to find if it is clockwise or anti-clockwise. This technique will prove to be invaluable when we deal with hidden line and surface algorithms later in this book. Listing 7.8 allows us to find whether or not three ordered two-dimensional vertices form an anti-clockwise triangle.

Example 7.8
Why is the polygon given in example 3.4 anti-clockwise?

The vertices (considered in three dimensions) are $(1, 0, 0), (5, 2, 0), (4, 4, 0)$ and $(-2, 1, 0)$. The directions of the edges are $(4, 2, 0), (-1, 2, 0), (-6, -3, 0)$ and $(3, -1, 0)$. Then, since

Listing 7.8

```
100 REM orientation of 2-D triangle
110 DIM X(3): DIM Y(3)
120 LET J$ = "TYPE IN COORDINATES OF TRIANGLE "
130 FOR I = 1 TO 3
140 LET I$ = "VERTEX(" + STR$ I + ")= ("
150 INPUT (J$ + I$);X(I);",";Y(I);")"
160 PRINT AT 2 + 2*I,0;I$;X(I);",";Y(I);")"
170 NEXT I
180 PRINT AT 12,0;"THE TRIANGLE IS ";
188 REM (DX1,DY1) is 2-D direction vector joining point 1 to point 2.
189 REM (DX2,DY2) is 2-D direction vector joining point 2 to point 3.
190 LET DX1 = X(2) - X(1): LET DY1 = Y(2) - Y(1)
200 LET DX2 = X(3) - X(2): LET DY2 = Y(3) - Y(2)
209 REM use 3-D vector product to check on orientation of triangle.
210 IF DX1*DY2 - DX2*DY1 > 0 THEN PRINT "ANTI-";
220 PRINT "CLOCKWISE": STOP
```

$$\begin{aligned}(4,2,0)\times(-1,2,0) &= (0,0,10)\\(-1,2,0)\times(-6,-3,0) &= (0,0,15)\\(-6,-3,0)\times(3,-1,0) &= (0,0,15)\\(3,-1,0)\times(4,2,0) &= (0,0,10)\end{aligned}$$

are all positive, so the orientation of the polygon is anti-clockwise. But be careful, if you lose this consistent order for calculating the vector product you can get the wrong answer. For example

$(-6,-3,0)\times(4,2,0) = (0,0,0)$ – the lines are parallel!

or $(-1,2,0)\times(3,-1,0) = (0,0,-5)$ – we have taken the edges out of sequence

Complete Programs

I. Listing 7.1 (intersection of line and plane). Data required: a base vector (B(1), B(2), B(3)) and direction vector (D(1), D(2), D(3)) for the line, a normal (N(1), N(2), N(3)) and constant K for the plane. Try (1, 2, 3), (1, 1, –1), (1, 0, 1) and 2 respectively.

II. Listing 7.2 (intersection of two lines). Data required: a base and direction vectors for the two lines. (B(1), B(2), B(3)) and (D(1), D(2), D(3)), and (C(1), C(2), C(3)) and (E(1), E(2), E(3)). Try (1, 2, 3), (1, 1, –1), and (–1, 1, 3), (1, 0, 1).

III. Listings 7.3 and 7.4 (main program, 'vecprod' and 'dotprod'). Data required: two vectors (L(1), L(2), L(3)) and (M(1), M(2), M(3)). Try (1, 2, 3), (1, 1, –1).

IV. Listings 7.5 ('inv') and 7.6 (intersection of three planes). Data required: normal (N(I, 1), N(I, 2), N(I, 3)) and constant K(I) for the three planes, $1 \leqslant I \leqslant 3$. Try (1, 2, 3), 0, (1, 1, −1), 1, (1, 0, 1), 2.

V. Listings 7.5 ('inv') and 7.7 (intersection of two planes). Data required: normal (N(I, 1), N(I, 2), N(I, 3)) and constant K(I) for the two planes, $1 \leqslant I \leqslant 2$. Try (1, 2, 3), 0, (1, 1, −1), 1.

IV. Listing 7.8 (orientation of 2-D triangle). Data required: the vertices (X(I), Y(I)), $1 \leqslant I \leqslant 3$. Try (1, 2), (2, 3) and (−1, 1).

8 *Matrix Representation of Transformations on Three-dimensional Space*

In chapter 4 we saw the need for transforming objects in two-dimensional space. When we draw three-dimensional pictures there will be many times when we need to make the equivalent linear transformations on three-dimensional space. As in the lower dimension, there are three basic types of transformation: translation, scaling and rotation. We shall represent transformations as square matrices (now they will be 4 × 4). A general point in space relative to a fixed coordinate triad, the row vector (x, y, z), must be considered as a four-rowed column vector

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

All the operations on matrices (addition, scalar multiple, transpose, premultiplication of a column vector and matrix product) that we saw in chapter 4 are easily extended to cope with 4 × 4 matrices and column vectors by simply changing the upper bound of the index ranges from 3 to 4. In this way we can generate a routine 'mult3' for multiplying two 4 × 4 matrices together. It is exactly equivalent to routine 'mult2' in the two-dimensional case, and for the very same reasons. The routine multiplies matrix A by matrix R giving matrix B, which is then copied into R. We also need the routine 'idR3', which sets R to the identity matrix (see listing 8.1).

Consider the case of a general linear transformation on points in three-dimensional space. A point (x, y, z) – 'before' – is transformed into (x', y', z') – 'after' – according to three *linear equations*

$$x' = A_{11} \times x + A_{12} \times y + A_{13} \times z + A_{14}$$
$$y' = A_{21} \times x + A_{22} \times y + A_{23} \times z + A_{24}$$
$$z' = A_{31} \times x + A_{32} \times y + A_{33} \times z + A_{34}$$

and as usual we add the extra equation

Listing 8.1

```
9100 REM mult3
9101 REM IN  : A(4,4),R(4,4)
9102 REM OUT : R(4,4)
9110 FOR I = 1 TO 4
9120 FOR J = 1 TO 4
9130 LET AR = 0
9140 FOR K = 1 TO 4
9150 LET AR = AR + A(I,K)*R(K,J)
9160 NEXT K
9170 LET B(I,J) = AR
9180 NEXT J
9190 NEXT I
9200 FOR I = 1 TO 4
9210 FOR J = 1 TO 4
9220 LET R(I,J) = B(I,J)
9230 NEXT J
9240 NEXT I
9250 RETURN

9300 REM idR3
9302 REM OUT : R(4,4)
9310 FOR I = 1 TO 4
9320 FOR J = 1 TO 4
9330 LET R(I,J) = 0
9340 NEXT J
9350 LET R(I,I) = 1
9360 NEXT I
9370 RETURN
```

$$1 = A_{41} \times x + A_{42} \times y + A_{43} \times z + A_{44}$$

which if it is to be true for all x, y and z means $A_{41} = A_{42} = A_{43} = 0$ and $A_{44} = 1$.

Then the equations may be written as a matrix equation where a column vector representing the 'after' point is the product of a matrix and the 'before' column vector

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

So if we store the transformation as a matrix, we can transform every required point by considering it as a column vector and *premultiplying* it by a transformation matrix. As before, transformations may be combined simply by obeying the sequence of transformations in order. If their equivalent matrices are A, B, C,. . ., L, M, N, then the matrix equivalent to the combination is $N \times M \times L \times \ldots \times C \times B \times A$. Remember the order. Since we are premultiplying a column vector, then the first transformation appears on the right of the matrix product and the last on the left.

Translation

Every point to be transformed is moved by a vector (TX, TY, TZ) say. This produces the following equations relating the 'before' and 'after' coordinates

$$\begin{aligned} x' &= 1 \times x + 0 \times y + 0 \times z + \mathrm{TX} \\ y' &= 0 \times x + 1 \times y + 0 \times z + \mathrm{TY} \\ z' &= 0 \times x + 0 \times y + 1 \times z + \mathrm{TZ} \end{aligned}$$

so that the matrix describing the translation is

$$\begin{pmatrix} 1 & 0 & 0 & \mathrm{TX} \\ 0 & 1 & 0 & \mathrm{TY} \\ 0 & 0 & 1 & \mathrm{TZ} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The routine 'tran3' for producing such a matrix A, given the parameters TX, TY and TZ is given in listing 8.2.

Listing 8.2

```
9000 REM tran3
9001 REM IN  : TX,TY,TZ
9002 REM OUT : A(4,4)
9010 FOR I = 1 TO 4
9020 FOR J = 1 TO 4
9030 LET A(I,J) = 0
9040 NEXT J
9050 LET A(I,I) = 1
9060 NEXT I
9070 LET A(1,4) = TX: LET A(2,4) = TY: LET A(3,4) = TZ
9080 RETURN
```

Scaling

The x-coordinate of every point to be transformed is scaled by a factor SX, the y-coordinate by SY and the z-coordinate by SZ, thus

$$\begin{aligned} x' &= \mathrm{SX} \times x + 0 \times y + 0 \times z + 0 \\ y' &= 0 \times x + \mathrm{SY} \times y + 0 \times z + 0 \\ z' &= 0 \times x + 0 \times y + \mathrm{SZ} \times z + 0 \end{aligned}$$

giving the matrix

$$\begin{pmatrix} \mathrm{SX} & 0 & 0 & 0 \\ 0 & \mathrm{SY} & 0 & 0 \\ 0 & 0 & \mathrm{SZ} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Usually the scaling values are positive, but if any of the values are negative then this leads to a reflection as well as (possibly) scaling. For example, if SX = −1 and SY = SZ = 1 then points are reflected in the y/z plane through the origin. A routine 'scale3' to produce such a scaling matrix A given SX, SY and SZ is given in listing 8.3.

Listing 8.3

```
8900 REM scale3
8901 REM IN  : SX,SY,SZ
8902 REM OUT : A(4,4)
8910 FOR I = 1 TO 4
8920 FOR J = 1 TO 4
8930 LET A(I,J) = 0
8940 NEXT J
8950 NEXT I
8960 LET A(1,1) = SX: LET A(2,2) = SY: LET A(3,3) = SZ
8970 LET A(4,4) = 1
8980 RETURN
```

Rotation about a Coordinate Axis

In order to consider the rotation about a general axis $\boldsymbol{p} + \mu\boldsymbol{q}$ by a given angle, it is first necessary to simplify the problem by considering rotation about one of the coordinate axes.

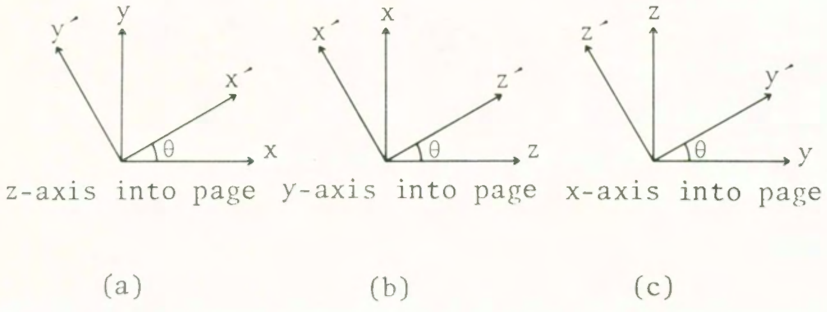


Figure 8.1

(a) Rotation by an angle θ about the x-axis

Referring to figure 8.1c, the axis of rotation is perpendicular to the page (the positive x-axis being into the page), and since we are using left-handed axes the figure shows the point (x', y', z') resulting from the transformation of an arbitrar point (x, y, z). We see that the rotation actually reduces to a two-dimensional rotation in the y/z plane passing through the point; that is, after the rotation the x-coordinate remains unchanged. Using the ideas explained in chapter 4 we have the following equations

$$\begin{aligned}x' &= x\\ y' &= \cos\theta \times y - \sin\theta \times z\\ z' &= \sin\theta \times y + \cos\theta \times z\end{aligned}$$

and thus the matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0\\ 0 & \cos\theta & -\sin\theta & 0\\ 0 & \sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

(b) Rotation by an angle θ about the y-axis

Referring to figure 8.1b, we now have the positive y-axis into the page and, because of the left-handedness of the axes, the positive z-axis is horizontal and to the right of the origin while the positive x-axis is above the origin. This leads us to the equations

$$\begin{aligned}x' &= \sin\theta \times z + \cos\theta \times x\\ y' &= y\\ z' &= \cos\theta \times z - \sin\theta \times x\end{aligned}$$

which gives the matrix

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

(c) Rotation by an angle θ about the z-axis

Referring to figure 8.1c we get the equations

$$\begin{aligned}x' &= \cos\theta \times x - \sin\theta \times y\\ y' &= \sin\theta \times x + \cos\theta \times y\\ z' &= z\end{aligned}$$

and the matrix

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

A subprogram 'rot3' to produce such a matrix A, given the angle THETA and the axis number AXIS (AXIS = 1 for the x-axis, AXIS = 2 for the y-axis and AXIS = 3 for the z-axis) is given in listing 8.4.

Listing 8.4

```
8600 REM rot3
8601 REM IN  : THETA,AXIS
8602 REM OUT : A(4,4)
8610 FOR I = 1 TO 4
8620 FOR J = 1 TO 4
8630 LET A(I,J) = 0
8640 NEXT J
8650 NEXT I
8660 LET A(4,4) = 1: LET A(AXIS,AXIS) = 1
8670 LET AX1 = AXIS + 1: IF AX1 = 4 THEN LET AX1 = 1
8680 LET AX2 = AX1 + 1: IF AX2 = 4 THEN LET AX2 = 1
8690 LET CT = COS THETA :LET ST = SIN THETA
8700 LET A(AX1,AX1) = CT: LET A(AX2,AX2) = CT
8710 LET A(AX1,AX2) = -ST: LET A(AX2,AX1) = ST
8720 RETURN
```

Inverse Transformations

Before we can consider the general rotation transformation, it is necessary to look at inverse transformations. An inverse transformation returns the points transformed by a given transformation back to their original position. If a transformation is represented by a matrix A, then the inverse transformation is given by matrix A^{-1}, the inverse of A. There is no need to explicitly calculate the inverse of a matrix using such techniques as the Adjoint Method (listing 7.5): we can use listings 8.2, 8.3 and 8.4 with parameters derived from the parameters of the original transformation

(1) A translation by (TX, TY, TZ) is inverted with a translation by (–TX, –TY, –TZ);
(2) a scaling by SX, SY and SZ is inverted with a scaling by 1/SX, 1/SY and 1/SZ;
(3) a rotation by an angle θ about a given axis is inverted with a rotation by an angle $-\theta$ about the same axis;
(4) if the transformation matrix is the product of a number of translation, scaling and rotation matrices $A \times B \times C \times \ldots \times L \times M \times N$, then the inverse transformation is

$$N^{-1} \times M^{-1} \times L^{-1} \times \ldots \times C^{-1} \times B^{-1} \times A^{-1}$$

Rotation of Points by an Angle γ about a General Axis $\boldsymbol{p} + \mu\boldsymbol{q}$

Assume $\boldsymbol{p} \equiv (\mathrm{PX}, \mathrm{PY}, \mathrm{PZ})$ and $\boldsymbol{q} \equiv (\mathrm{QX}, \mathrm{QY}, \mathrm{QZ})$. We break down the task into a number of subtasks

(a) We translate all of space so that the axis of rotation goes through the origin. This is achieved by adding a vector $-\boldsymbol{p}$ to every point in space with a matrix F say, which is generated by a call to 'tran3' with parameters TX = −PX, TY = −PY and TZ = −PZ. The inverse matrix F^{-1} will be needed later and is found by a call to 'tran3' with parameters PX, PY and PZ. After this transformation the axis of rotation is the line $\boldsymbol{O} + \mu\boldsymbol{q}$ passing through the origin.

$$F = \begin{pmatrix} 1 & 0 & 0 & -\mathrm{PX} \\ 0 & 1 & 0 & -\mathrm{PY} \\ 0 & 0 & 1 & -\mathrm{PZ} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad F^{-1} = \begin{pmatrix} 1 & 0 & 0 & \mathrm{PX} \\ 0 & 1 & 0 & \mathrm{PY} \\ 0 & 0 & 1 & \mathrm{PZ} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) We then rotate space about the z-axis by an angle $-\alpha$, where (ALPHA =) $\alpha = \tan^{-1}$ (QY/QX), given by the matrix G. The matrix is generated by a call to 'rot3', setting THETA = −ALPHA and AXIS = 3, and the inverse matrix G^{-1} by a call to 'rot3' with THETA = ALPHA and AXIS = 3. At this stage the axis of rotation is a line lying in the x/z plane passing through the point $(v, 0, \mathrm{QZ})$.

$$G = \frac{1}{v}\begin{pmatrix} \mathrm{QX} & \mathrm{QY} & 0 & 0 \\ -\mathrm{QY} & \mathrm{QX} & 0 & 0 \\ 0 & 0 & v & 0 \\ 0 & 0 & 0 & v \end{pmatrix} \quad G^{-1} = \frac{1}{v}\begin{pmatrix} \mathrm{QX} & -\mathrm{QY} & 0 & 0 \\ \mathrm{QY} & \mathrm{QX} & 0 & 0 \\ 0 & 0 & v & 0 \\ 0 & 0 & 0 & v \end{pmatrix}$$

where v is the positive number given by $v^2 = \mathrm{QX}^2 + \mathrm{QY}^2$.

(c) We now rotate space about the y-axis by an angle $-\beta$, where (BETA =) $\beta = \tan^{-1}$ (v/QZ), given by the matrix H. This matrix is generated by a call to 'rot3' with parameters AXIS = 2 and THETA = −BETA, and the inverse matrix H^{-1} by a call to 'rot3' with parameters AXIS = 2 and THETA = BETA.

$$H = \frac{1}{w}\begin{pmatrix} \mathrm{QZ} & 0 & -v & 0 \\ 0 & w & 0 & 0 \\ v & 0 & \mathrm{QZ} & 0 \\ 0 & 0 & 0 & w \end{pmatrix} \quad H^{-1} = \frac{1}{w}\begin{pmatrix} \mathrm{QZ} & 0 & v & 0 \\ 0 & w & 0 & 0 \\ -v & 0 & \mathrm{QZ} & 0 \\ 0 & 0 & 0 & w \end{pmatrix}$$

where w is the positive number given by $w^2 = v^2 + \mathrm{QZ}^2 = \mathrm{QX}^2 + \mathrm{QY}^2 + \mathrm{QZ}^2$. So the point $(v, 0, \mathrm{QZ})$ is transformed to $(0, 0, w)$, hence the axis of rotation is along the z-axis.

(d) We can now rotate space by an angle γ (GAMMA) about the axis of rotation using matrix W generated by 'rot3' (with AXIS = 3 and THETA = GAMMA).

$$W = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 1 \\ \sin\gamma & \cos\gamma & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(e) We need to return the axis of rotation to its original position so we multiply H^{-1}, G^{-1} and finally F^{-1}.

Thus the final matrix P that rotates space by the angle γ about the axis $\boldsymbol{p} + \mu\boldsymbol{q}$ is $P = F^{-1} \times G^{-1} \times H^{-1} \times W \times H \times G \times F$. Naturally some of these matrices may reduce to the identity matrix in some special cases, and can be ignored. For example, if the axis of rotation goes through the origin then F and F^{-1} are identical to the identity matrix I, and can be ignored.

So it is possible to write a special routine 'genrot' (listing 8.5) that achieves this rotation and returns the required matrix P given GAMMA, (PX, PY, PZ) and (QX, QY, QZ).

Listing 8.5

```
5800 REM genrot
5801 REM IN  : PX,PY,PZ,QX,QY,QZ,GAMMA,R(4,4)
5802 REM OUT : R(4,4)
5809 REM place origin on axis of rotation.
5810 LET TX = -PX: LET TY = -PY: LET TZ = -PZ: GO SUB tran3: GO SUB mult3
5819 REM rotate axis of rotation into x/z plane.
5820 LET AX = QX: LET AY = QY: GO SUB angle
5830 LET ALPHA = THETA: LET THETA = -THETA: LET AXIS = 3: GO SUB rot3
     : GO SUB mult3
5839 REM rotate axis of rotation onto z-axis.
5840 LET AX = QZ: LET AY = SQR (QX*QX + QY*QY): GO SUB angle
5850 LET BETA = THETA: LET THETA = -THETA: LET AXIS = 2: GO SUB rot3
     : GO SUB mult3
5859 REM rotate by GAMMA about axis of rotation.
5860 LET AXIS = 3: LET THETA = GAMMA: GO SUB rot3: GO SUB mult3
5869 REM replace axis back to original position.
5870 LET AXIS = 2: LET THETA = BETA: GO SUB rot3: GO SUB mult3
5880 LET AXIS = 3: LET THETA = ALPHA: GO SUB rot3: GO SUB mult3
5890 LET TX = PX: LET TY = PY: LET TZ = PZ: GO SUB tran3: GO SUB mult3
5900 RETURN
```

Example 8.1

What happens to the points (0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1) and (1, 1, 1) if space is rotated by $\pi/4$ radians about an axis $(1, 0, 1) + \mu(3, 4, 5)$.

Using the above theory we note that

$$F = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad F^{-1} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$G = \frac{1}{5}\begin{pmatrix} 3 & 4 & 0 & 0 \\ -4 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 5 \end{pmatrix} \qquad G^{-1} = \frac{1}{5}\begin{pmatrix} 3 & -4 & 0 & 0 \\ 4 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 5 \end{pmatrix}$$

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & \sqrt{2} & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & \sqrt{2} \end{pmatrix} \qquad H^{-1} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & \sqrt{2} & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & \sqrt{2} \end{pmatrix}$$

$$W = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{2} & 0 \\ 0 & 0 & 0 & \sqrt{2} \end{pmatrix} \quad \text{and}$$

$$P = \frac{1}{50\sqrt{2}}\begin{pmatrix} 41 + 9\sqrt{2} & -12 - 13\sqrt{2} & -15 + 35\sqrt{2} & -26 + 6\sqrt{2} \\ -12 + 37\sqrt{2} & 34 + 16\sqrt{2} & -20 + 5\sqrt{2} & 32 - 42\sqrt{2} \\ -15 - 5\sqrt{2} & -20 + 35\sqrt{2} & 25 + 25\sqrt{2} & -10 + 30\sqrt{2} \\ 0 & 0 & 0 & 50\sqrt{2} \end{pmatrix}$$

where $P = F^{-1} \times G^{-1} \times H^{-1} \times W \times H \times G \times F$ is the matrix representation of the required transformation. Premultiplying the column vectors equivalent to $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ and $(1, 1, 1)$ by P and changing the resulting column vectors back into row form and taking out a factor $1/50\sqrt{2}$ gives the co-ordinates $(-26 + 6\sqrt{2}, 32 - 42\sqrt{2}, -10 + 30\sqrt{2})$, $(15 + 15\sqrt{2}, 20 - 5\sqrt{2}, -25 + 25\sqrt{2})$, $(-38 - 7\sqrt{2}, 66 - 26\sqrt{2}, -30 + 65\sqrt{2})$, $(-41 + 41\sqrt{2}, 12 - 37\sqrt{2}, 15 + 55\sqrt{2})$ and $(-12 + 37\sqrt{2}, 34 + 16\sqrt{2}, -20 + 85\sqrt{2})$ respectively. Naturally translating and rotating space should leave relative positions unchanged; in particular the angles between direction vectors should be unchanged (the same cannot be said about the scaling transformation, which in general does alter relative positions). In the original system the three relative positions from $(0, 0, 0)$ to $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$ respectively, are mutually perpendicular (that is, the dot product of pairs of these directions should be zero). The dot product of the directions in the transformed system should also be zero: the three directional vectors (with $1/50\sqrt{2}$ factored out) are $(41 + 9\sqrt{2}, -12 + 37\sqrt{2}, -15 - 5\sqrt{2})$, $(-12 - 13\sqrt{2}, 34 + 16\sqrt{2}, -20 + 35\sqrt{2})$ and $(-15 + 35\sqrt{2}, -20 + 5\sqrt{2}, 25 + 25\sqrt{2})$, and the dot product of any pair is zero.

Similarly the dot product of the direction vector from the origin to $(1, 1, 1)$ in the original system, taken with any of the original directions above, give the same value $(= 1)$. This is also true in the transformed system: the fourth direction is $(14 + 31\sqrt{2}, 2 + 58\sqrt{2}, -10 + 55\sqrt{2})$, and when we take the dot product with each of the three direction vectors above we get the value 5000, which, when we take into account the factor $(1/50\sqrt{2})^2$, gives the value 1.

A program that reads in the axis of rotation $(PX, PY, PZ) + \mu(QX, QY, QZ)$

and the angle GAMMA, and rotates any point (XX, YY, ZZ) about this axis by angle GAMMA is given in listing 8.6.

Listing 8.6

```
100 REM rotation of a point about a given axis
110 DIM A(4,4): DIM B(4,4): DIM R(4,4)
120 DIM P(3): DIM A$(8)
130 INPUT "BASE VECTOR OF AXIS ","(";PX;",";PY;",";PZ;")"
140 PRINT AT 1,0;"BASE VECTOR OF AXIS ","(";PX;",";PY;",";PZ;")"
150 INPUT "DIRECTION VECTOR OF AXIS ","(";QX;",";QY;",";QZ;")"
160 PRINT AT 4,0;"DIRECTION VECTOR OF AXIS ","(";QX;",";QY;",";QZ;")"
170 INPUT "ANGLE OF ROTATION ";GAMMA
180 PRINT AT 7,0;"ANGLE OF ROTATION ";GAMMA
190 LET mult3 = 9100: LET idR3 = 9300: LET rot3 = 8600
    : LET tran3 = 9000: LET angle = 8800: LET genrot = 5800
198 REM calculate R(4,4) for rotating point by angle GAMMA about an axis
199 REM with base vector (PX,PY,PZ) and direction vector (QX,QY,QZ).
200 GO SUB idR3: GO SUB genrot
210 PRINT AT 14,0;"BECOMES"
219 REM input and transform point (XX,YY,ZZ).
220 INPUT "POINT VECTOR","(";XX;",";YY;",";ZZ;")"
230 PRINT AT 12,0;"POINT (";XX;",";YY;",";ZZ;")",,
240 LET P(1) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
250 LET P(2) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
260 LET P(3) = XX*R(3,1) + YY*R(3,2) + ZZ*R(3,3) + R(3,4)
269 REM tidy up output.
270 PRINT AT 16,0;"(";
280 FOR I = 1 TO 3: LET A$ = STR$ P(I): FOR J = 1 TO 8
290 IF A$(J) <> " " THEN PRINT A$(J);
300 NEXT J: IF I <> 3 THEN PRINT ",";
310 NEXT I: PRINT ")",,
320 GO TO 220
```

Exercise 8.1

Experiment with these ideas. You can always make a check on your final transformation matrix by considering simple values as above, and you can use the previous listings to check your answer. It is essential that you are confident in the use of matrices, and the best way to get this confidence is to experiment. You will make lots of arithmetic errors initially, but you will soon come to think of transformations in terms of their matrix representation, and this will greatly ease the study of drawing three-dimensional objects.

Exercise 8.2

As with the two-dimensional case, we note that the 'bottom row' of all transformation matrices is always (0, 0, 0, 1), and is of no real use in calculations. It is added only to form square matrices, which are necessary for the formal definition of matrix multiplication. We can adjust this definition, and that of the multiplication of a matrix and a column vector, so that instead we use only the top three rows of the 4 X 4 matrices (in chapter 4 we used the top two rows of 3 X 3 matrices in listings 4.2a, 4.3a, 4.4a and 4.5a). Change listings 8.1, 8.2, 8.3 and 8.4 accordingly.

Exercise 8.3

You will have noticed that the routine ‘rot3’ is usually called with THETA generated by ‘angle’, which uses values AX and AY as input parameters. ‘rot3’ calculates the cosine and sine of angle THETA, but we know that these are $AX/\sqrt{(AX^2 + AY^2)}$ and $AY/\sqrt{(AX^2 + AY^2)}$ respectively. Write another rotation routine ‘rotxy’ that calculates the rotation matrix directly from AX and AY without resorting to ‘angle’.

Also we note that the first stage of the ‘rot’, ‘tran’, ‘scale’ and ‘idR’ routines consists of clearing an array. This can be achieved more efficiently on the Spectrum by reDIMensioning the array. Furthermore it is often faster to explicitly assign values rather than calculate them inside FOR. . .NEXT loops.

Exercise 8.4

Again in chapter 4 we noted that some writers use row rather than column vectors, and postmultiply rather than premultiply. We decided against this interpretation, so that the matrix of a transformation corresponds directly with the coefficients of the transformation equations. In this other interpretation it is the transpose of the matrix that is identical to the coefficients. It is useful to be aware of this other method, so use it to rewrite all the programs given in this chapter (and the remainder of this book). Remember though, it is not important which method you finally decide to use, as long as you are consistent. We use the column vector notation because we have found that it causes less confusion in the early stages of learning the subject!

Comple Programs

I. All the listings in this chapter, 8.1 (‘mult3’ and ‘idR3’), 8.2 (‘tran3’), 8.3 (‘scale3’), 8.4 (‘rot3’), 8.5 (‘genrot’), 8.6 (main program) and listing 3.4 (‘angle’). Required data: base vector (PX, PY, PZ), direction vector (QX, QY, QZ) of the axis of rotation and the angle GAMMA, together with any number of three-dimensional coordinates (XX, YY, ZZ). Try (0, 0, 0), (1, 1, 1) and $\pi/4$, and points (1, 0, 1), (1, 1, 1), (1, 2, 3).

9 *Orthographic Projections*

We may now address the problem of drawing views of three-dimensional objects on our (necessarily) two-dimensional graphics screen. The simple method we describe here is a direct generalisation of the method introduced in chapter 4 for two-dimensional objects. Again it involves the use of (up to) three *positions*. To illustrate these ideas we first give a brief outline, and then expand on this using numerous pictorial and numerical examples. We start by defining an arbitrary but fixed triad of axes in space that we call the ABSOLUTE system. Then, as in the two-dimensional case, we consider three positions: (1) the SETUP, (2) the ACTUAL and (3) the OBSERVED position.

(1) The SETUP Position

Most scenes will be composed of simple objects (for example, cube(s), see example 9.1) that are set at a peculiar position and orientation in space. It is very inefficient to calculate by hand the complicated coordinates of every vertex of these objects and input them into the program. Instead we look at each object in turn and initially define it in an elementary way relative to the ABSOLUTE triad, usually setting it about the origin. The information required will be that of vertices (x-coordinate, y-coordinate and z-coordinate), and perhaps lines (that join pairs of vertices) or (later when we consider hidden line and hidden surface algorithms) facets, which are polygonal planar areas bounded by the above-mentioned lines. This elementary definition of the object is called its SETUP position. We could have other information also, such as the colour of the object.

(2) The ACTUAL position

We use the matrix techniques of the last chapter to generate a matrix that will move the object from its SETUP position to its required ACTUAL position relative to the ABSOLUTE axes. We shall call this SETUP to ACTUAL matrix P.

(3) The OBSERVED Position

Viewing an object in three-dimensional space naturally involves an observer (the eye – and note only one eye!) placed at a position (EX, EY, EZ) relative to the ABSOLUTE axes looking in a fixed direction: this direction of view can be uniquely determined by any other point on the line of sight, (DX, DY, DZ) say. The head can also be tilted, but more of that later. What the eye sees when looking at a three-dimensional object is a projection of the vertices, lines (and facets) of the object on to a (two-dimensional) *view plane* that is normal to the line of sight. In order to calculate such projections we must standardise our approach. We use matrix methods to transform all the points in space so that the eye is placed at the origin, and the line of sight is along the positive z-axis. This is the OBSERVED position, and the matrix that transforms the ACTUAL to OBSERVED position is called Q throughout this book. The method for calculating Q will be dealt with in detail later, but for the time being we assume that the eye is already at the origin looking along the z-axis: so in this simple case Q is the identity matrix.

When all the points in space have been moved into this OBSERVED position, we note that the view plane is now parallel to the x/y plane through the origin. Having moved the eye into the correct position, we are now ready to project the object on to the view plane. But note, as yet we have neither defined the position of the view plane (we have only its normal), nor have we described the type of projection of three-dimensional space on to the plane. These two requirements are closely related. In this book we shall consider two possible projections: in a later chapter we shall deal with the *perspective projection*, but first we introduce the simplest projection, the *orthographic*.

The Orthographic Projection

Nothing could be simpler. In the orthographic projection we can set the view plane to be *any* plane with normal vector along the line of sight. When transformed into the OBSERVED position, the view plane will be any plane parallel to the x/y plane given by the equation $z = 0$. For simplicity we take the x/y plane through the origin. The vertices of the object are projected on to the view plane by the simple expedient of setting their z-coordinates to zero. Thus any two different points in the OBSERVED position, (x, y, z) and (x, y, z') say (where $z \neq z'$), are projected on to the same point $(x, y, 0)$ on the view plane. Then we identify the x/y values on the plane with points in the graphics screen coordinate system (usually centred on the screen) using the methods of chapter 2. Once the vertices have been projected on to the view plane and then on to the screen, we can construct the projection of lines and facets. These are related to the projected vertices in exactly the same way as the original lines and facets are related to the original vertices.

Before considering in detail the general case where the eye and direction of view are arbitrarily positioned, we take an elementary example to demonstrate the orthographic projection.

Example 9.1

Use the above ideas to draw an orthographic projection of a cube. Figures such as those in figure 9.1 are called *wire diagrams* or *skeletons* (for obvious reasons).

In the SETUP position the cube may be thought to consist of eight vertices (1, 1, 1), (1, 1, –1), (1, –1, –1), (1, –1, 1), (–1, 1, 1), (–1, 1, –1), (–1, –1, –1) and (–1, –1, 1): vertices labelled numerically 1 to 8. The twelve lines that form the wire cube join vertices 1 to 2, 2 to 3, 3 to 4, 4 to 1; 5 to 6, 6 to 7, 7 to 8, 8 to 1; 1 to 5, 2 to 6, 3 to 7, 4 to 8.

Figure 9.1a shows the simplest possible example of an orthographic projection of the cube, where even the SETUP to ACTUAL matrix is the identity matrix; that is, the cube stays in its SETUP position. We get a square: pairs of parallel lines from the front and back of the cube project into the same line on the screen. We put a "+" in these diagrams to show the position of the z-axis in the OBSERVED position (into the screen).

Figure 9.1b shows the same cube drawn after the following three transformations place it in its ACTUAL position.

(a) Rotate the cube by an angle $\alpha = -0.927295218$ radian about the z-axis – matrix A. This example is contrived so that $\cos \alpha = 3/5$ and $\sin \alpha = -4/5$, ensuring that the rotation matrices consist of uncomplicated elements.
(b) Translate by the vector $(-1, 0, 0)$ – matrix B.
(c) Rotate by an angle $-\alpha$ about the y-axis – matrix C.

The SETUP to ACTUAL matrix is thus $P = C \times B \times A$, where

$$A = \begin{pmatrix} 3/5 & 4/5 & 0 & 0 \\ -4/5 & 3/5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 3/5 & 0 & 4/5 & 0 \\ 0 & 1 & 0 & 0 \\ -4/5 & 0 & 3/5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and P is given by

$$P = \frac{1}{25}\begin{pmatrix} 9 & 12 & 20 & -15 \\ -20 & 15 & 0 & 0 \\ -12 & -16 & 15 & 20 \\ 0 & 0 & 0 & 25 \end{pmatrix}$$

So the above eight vertex coordinate triples in the SETUP position are transformed into the following eight ACTUAL coordinate triples: (26/25, –5/25, 7/25), (–14/25, –5/25, –23/25), (–38/25, –35/25, 9/25), (2/25, –35/25, 39/25), (8/25, 35/25, 31/25), (–32/25, 35/25, 1/25), (–56/25, 5/25, 33/25), (–16/25, 5/25, 63/25).

For example (1, 1, 1) is transformed to (26/25, –5/25, 7/25) because

$$\frac{1}{25}\begin{pmatrix} 9 & 12 & 20 & -15 \\ -20 & 15 & 0 & 0 \\ -12 & -16 & 15 & 20 \\ 0 & 0 & 0 & 25 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \frac{1}{25}\begin{pmatrix} 26 \\ -5 \\ 7 \\ 25 \end{pmatrix}$$

Since the ACTUAL to OBSERVED matrix Q is the identity matrix, the projected coordinates on the view plane are thus (26/25, –5/25), (–14/25, –5/25), (–38/25, –35/25), (2/25, –35/25), (8/25, 35/25), (–32/25, 35/25), (–56/25, 5/25), (–16/25, 5/25). We can place these points on the screen and join them with lines in the same order as they were defined in the SETUP cube.

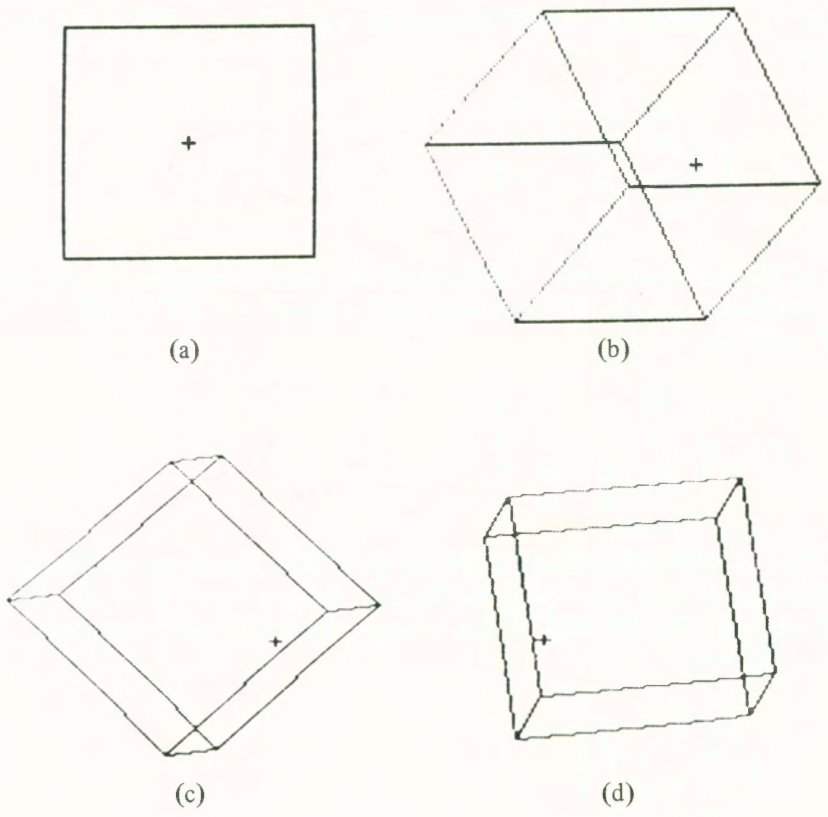


Figure 9.1

Construction of the ACTUAL to OBSERVED Matrix Q

We assume that the eye is at (EX, EY, EZ) relative to the ABSOLUTE axes, looking towards the point (DX, DY, DZ). The OBSERVED position is achieved in the following sequence of steps.

(1) A matrix D translates all the points in space by a vector (–DX, –DY, –DZ) so that now the eye is at (EX – DX, EY – DY, EZ – DZ) = (FX, FY, FZ) say, looking towards the origin.

$$D = \begin{pmatrix} 1 & 0 & 0 & -\mathrm{DX} \\ 0 & 1 & 0 & -\mathrm{DY} \\ 0 & 0 & 1 & -\mathrm{DZ} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(2) A matrix E changes (FX, FY, FZ) into $(r, 0, \mathrm{FZ})$ by rotating space by an angle $-\alpha$ where $\alpha = \tan^{-1}$ (FY/FX) about the z-axis. Here $r^2 = \mathrm{FX}^2 + \mathrm{FY}^2$ and $r > 0$.

$$E = \frac{1}{r}\begin{pmatrix} \mathrm{FX} & \mathrm{FY} & 0 & 0 \\ -\mathrm{FY} & \mathrm{FX} & 0 & 0 \\ 0 & 0 & r & 0 \\ 0 & 0 & 0 & r \end{pmatrix}$$

(3) A matrix F transforms $(r, 0, \mathrm{FZ})$ into $(0, 0, -s)$ by rotating space by an angle $\pi - \theta$ about the y-axis, where $\theta = \tan^{-1}$ (r/FZ). Here $s^2 = r^2 + \mathrm{FZ}^2 = \mathrm{FX}^2 + \mathrm{FY}^2 + \mathrm{FZ}^2$ and $s > 0$.

$$F = \frac{1}{s}\begin{pmatrix} -\mathrm{FZ} & 0 & r & 0 \\ 0 & s & 0 & 0 \\ -r & 0 & -\mathrm{FZ} & 0 \\ 0 & 0 & 0 & s \end{pmatrix}$$

(4) The transformation thus far places the eye at $(0, 0, -s)$ on the negative z-axis looking towards the origin and at the same distance from it (s) as (EX, EY, EZ) was from (DX, DY, DZ). We now generate a matrix G, which moves the eye to the origin.

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & s \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(5) If in example 9.1, we now premultiply $P = C \times B \times A$ by our first approximation to the ACTUAL to OBSERVED matrix Q (= $G \times F \times E \times D$) to find

the SETUP to OBSERVED matrix $R = Q \times P = G \times F \times E \times D \times C \times B \times A$, we draw figure 9.1c by orthographic projection. This view is not really satisfactory because the matrix Q places the cube at an arbitrary orientation within the view plane. It is much better to standardise our view, and one of the most popular ways is to *maintain the vertical*, that is a line that was vertical (or parallel to the y-axis) in its ACTUAL position remains vertical after transformation by Q into its OBSERVED position. Take the vertical line from (DX, DY, DZ) to (DX, DY + 1, DZ). Because of this peculiar construction, we note that intermediate matrix K ($F \times E \times D$) transforms this line into one joining (0, 0, 0) to (K(1,2), K(2,2), K(3,2)) = (p, q, r), say. So if we further rotate about the z-axis by an angle $\beta = \tan^{-1}$ (K(1, 2)/K(2, 2)) = $\tan^{-1} (p/q) = \tan^{-1}$ (–FY x FZ/ (s x FX)) using a matrix H, before multiplying by G, then the vertical is maintained

$$H = \frac{1}{t}\begin{pmatrix} q & -p & 0 & 0 \\ p & q & 0 & 0 \\ 0 & 0 & t & 0 \\ 0 & 0 & 0 & t \end{pmatrix} \quad \text{where} \quad t^2 = p^2 + q^2$$

and thus

$$H \times \begin{pmatrix} p \\ q \\ r \\ 1 \end{pmatrix} = \frac{1}{t}\begin{pmatrix} q & -p & 0 & 0 \\ p & q & 0 & 0 \\ 0 & 0 & t & 0 \\ 0 & 0 & 0 & t \end{pmatrix} \times \begin{pmatrix} p \\ q \\ r \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ t \\ r \\ 1 \end{pmatrix}$$

Thus the complete transformation (figure 9.1d) is achieved by the matrix $R = Q \times P = G \times H \times F \times E \times D \times C \times B \times A$, and the projection of the line joining points (DX, DY, DZ) to (DX, DY + 1, DZ) is the line joining (0, 0) to (0, t) on the screen; that is, the vertical. Matrix G does not affect the x/y values. Note that this technique works in all cases except where (EX, EY, EZ) is vertically above (DX, DY, DZ) to start with, and naturally maintaining the vertical makes no sense. The routine 'look3' (listing 9.1), given (EX, EY, EZ) and (DX, DY, DZ), generates the ACTUAL to OBSERVED matrix in the steps shown above, and at each step premultiplies the matrix R: so at the end of the process, R will hold its original matrix value premultiplied by Q. If we wish to store Q explicitly, then we need first to set R to the identity matrix (using 'idR3'), then call 'look3', and finally copy array R into array Q. Routine 'look3' can be radically reduced if we assume that the eye always looks at the origin (that is, DX = DY = DZ = 0). Furthermore with the orthographic projection the OBSERVED position of the eye need not be at the origin, it merely needs to be on the z-axis: again the routine can be cut down. We give the most general case, which will be essential for later perspective projections.

Listing 9.1

```
8200 REM look3
8210 INPUT "(EX,EY,EZ) ";EX;",";EY;",";EZ
8220 INPUT "(DX,DY,DZ) ";DX;",";DY;",";DZ
8229 REM move origin to (DX,DY,DZ).
8230 LET TX = -DX: LET TY = -DY: LET TZ = -DZ
8240 GO SUB tran3: GO SUB mult3
8249 REM move eye onto negative z-axis, looking at the origin.
8250 LET FX = EX - DX: LET FY = EY - DY: LET FZ = EZ - DZ
8260 LET AX = FX: LET AY = FY: GO SUB angle
8270 LET AXIS = 3: LET THETA = -THETA: GO SUB rot3: GO SUB mult3
8280 LET AX = FZ: LET AY = SQR (FX*FX + FY*FY): GO SUB angle
8290 LET AXIS = 2: LET THETA = PI - THETA: GO SUB rot3: GO SUB mult3
8299 REM maintain vertical.
8300 LET TX = 0: LET TY = 0: LET TZ = SQR (FX*FX + FY*FY + FZ*FZ)
8310 LET AX = TZ*FX: LET AY= -FY*FZ: GO SUB angle
8320 LET AXIS = 3: GO SUB rot3: GO SUB mult3
8329 REM move eye to the origin: space is now in the OBSERVED position.
8330 GO SUB tran3: GO SUB mult3
8340 RETURN
```

If required, we can extend this program to deal with the situation where the head is tilted through an angle γ from the vertical. This is achieved by further rotating space by $-\gamma$ about the z-axis. Thus matrix H should then rotate about the z-axis by an angle $\beta - \gamma$.

The construction of the ACTUAL to OBSERVED matrix is obviously independent of everything other than the position of the eye, line of sight and the tilt of the head. So if we wish to view a series of objects from the same position, we can store Q and use it repeatedly for placing each object.

How to Define an Object

It is now time to deal with the problem of representing objects to the computer. There is no definite solution, it really depends on what is being drawn and how it is projected. In this section we describe various ways of setting up a data-base to hold the information necessary for drawing any given scene, but make no comment on their usefulness. This is considered in the remainder of the book where we give examples to illustrate the value of particular methods in different situations. We shall be using arrays to hold large sets of data, and so naturally the amount of space given to arrays will depend on the amount of information required for a scene: be sure that when you declare these arrays there is enough space for all the information; if in doubt, overestimate your store requirements.
Vertices. We will always need to define vertices and other special reference points in a scene, and these we store as x, y and z-coordinates in arrays X, Y and Z respectively, assuming that if the total number is not known explicitly, then this value is calculated as NOV. So there must be space for not less than NOV values in each of the three arrays. These vertices may be in the SETUP, ACTUAL or OBSERVED position, it depends on the context of the problem. There will

also be situations (perspective in particular) when we need to store the *x*/*y*-coordinates of the projections of these NOV vertices – in arrays V and W. Naturally this is unnecessary in the case of an orthographic projection of points in the OBSERVED position since we can use the values already stored in the X and Y arrays. The choice of data-base really depends on the scene and type of projection.
Lines. We can store information on NOL (say) line segments in the two dimensional integer array L. The I^th^ line is defined by the integer indices (between 1 and NOV) of the two points at each end of the line – we store the indices in L(1, I) and L(2, I). The true coordinate values of the two points at each end of the line segment can be found from the X, Y and Z arrays.
Facets. A facet is a convex polygonal area on the surface of a three-dimensional object, and can be defined in a number of ways. Most facets will be triangular or quadrilateral, so we usually assume that no facet has greater than six sides to minimise waste of store. The NOF facets can be defined in terms of the indices of the vertices at their corners in array F; F(I, J) is the index of the I^th^ vertex on the J^th^ facet. Naturally if the facet is not hexagonal then some of the values are *garbage* so we need to store array H, the number of vertices/edges on each facet. We can also store C, the integer colour code (if any) for each facet; but be careful, the Spectrum allows only two colours in any one character block. Another method is to store the facet in terms of the indices of the lines in the object in array F, which would thus refer to array L; F(I, J) would now be the index of the I^th^ line on the edge of the J^th^ facet. There are many other methods for representing these, and other elements of a three-dimensional object: choose the one most suitable to your particular situation.

Construction Routines and the 'Building Block' Method

For any required object we define a *construction routine* that needs, as parameters, a matrix *R* to move vertices into position and any other information about the size of the object (if the object is to be stored in the SETUP position then naturally no matrix is needed). The routine can then define the vertices, lines, facets or any other elements of the object, and use the matrix *R* to move the vertices of the object into the required position. Depending on the context of the program, the routine can then either draw the object, or extend a database containing this information. We shall give examples of both methods.

We can construct a scene containing a number of similar objects (so the data will be in either the ACTUAL or the OBSERVED position). There is no need to produce a new construction routine for each occurrence of the object, all we do each time is calculate a new SETUP to ACTUAL matrix *P*, and enter it (for the ACTUAL position) or $Q \times P$ (for the OBSERVED position) into the same routine. Naturally we require one new routine for each different type of object.

The complete scene is achieved by the execution of a main program (listing 9.2), which declares all the subroutine labels, then prepares the graphics screen using input values of HORIZ and VERT and finally calls a routine 'scene3' that organises the objects in space and then draws them. The main program below will be used in all the three-dimensional graphics programs that follow, so do not alter it without very good reason.

Listing 9.2

```
100 REM main program
110 LET start = 9700: LET setorigin = 9600: LET moveto = 9500
    : LET lineto = 9400: LET clip = 8400
120 LET rot3 = 8600: LET angle = 8800: LET scale3 = 8900: LET tran3 = 9000
    : LET mult3 = 9100: LET idR3 = 9300
130 LET scene3 = 6000: LET look3 = 8200
139 REM dimension and centre the graphics area.
140 INPUT "HORIZ ",HORIZ,"VERT ",VERT
150 GO SUB start
160 LET XMOVE = HORIZ*0.5: LET YMOVE = VERT*0.5
170 GO SUB setorigin
179 REM set the scene.
180 GO SUB scene3
190 STOP
```

'scene3' declares all the arrays that are required for storing information about a scene, together with matrices A, B, R and (perhaps) Q for moving objects into position. Any other subroutine labels unique to this scene are also declared. If required the values of NOV and NOL (or NOF) are initialised, and these will be updated in later construction routines. For each individual object (a 'block), 'scene3' must calculate a matrix P that moves this block into the ACTUAL position, and then call the construction routine using the correct matrix R (perhaps SETUP to ACTUAL or SETUP to OBSERVED). All the blocks finally construct the finished scene. Sometimes the drawing of the projection is done inside the construction routine, or it can be elsewhere in other routines specifically designed for special forms of drawing (as in hidden line and hidden surface pictures): it depends on what is being drawn and what is required of the view. As usual, because of the restriction of not passing array parameters into subroutines, we do not normally explicitly generate P and Q, but rely instead on updating matrix R. If we require the ACTUAL to OBSERVED matrix then this routine calls 'look3'. Should we need to store Q then we must first call 'idR3', which sets matrix R to the identity; remember all matrix operations are done via matrices A and R, using matrix B to hold intermediate values.

Always use the clipping version of 'lineto' in your programs. It is so easy to choose values of HORIZ and VERT that are too small, with the result that part of the object goes outside the graphics area. Without the 'clip' routine the program will fail. There is a positive reason, however, for making these values small: it enables us to zoom in on a small area of a scene, drawing it very large, and all the exterior lines will be clipped away.

Our first example of this method is listing 9.3, which is the 'scene3' routine needed to construct a picture of a single cube as shown in figure 9.1d. The scene can be viewed from any position with the vertical maintained. We have also a construction routine 'cube' (listing 9.4) that generates the data for a cube with sides of length 2. It places the vertices, eight sets of coordinate triples, in arrays X, Y and Z. There is no need to store the lines of the cube explicitly, we get the information from a DATA statement and draw the lines straight away. The data for figure 9.1d are HORIZ = 6, VERT = 4, (EX, EY, EZ) = (–2, 2, 2) and (DX, DY, DZ) = (–1, 0, 0).

Listing 9.3

```
6000 REM scene3/example 9.1
6010 DIM X(8): DIM Y(8): DIM Z(8)
6020 DIM A(4,4): DIM B(4,4): DIM R(4,4)
6030 LET cube = 6500
6039 REM calculate the SETUP to ACTUAL matrix R.
6040 GO SUB idR3
6050 LET THETA = -0.92729522: LET AXIS = 3: GO SUB rot3: GO SUB mult3
6060 LET TX = -1: LET TY = 0: LET TZ = 0: GO SUB tran3: GO SUB mult3
6070 LET THETA = -THETA: LET AXIS = 2: GO SUB rot3: GO SUB mult3
6079 REM change R: premultiply it by the ACTUAL to OBSERVED matrix.
6080 GO SUB look3
6089 REM call construction routine draw the cube.
6090 GO SUB cube
6100 RETURN
```

Listing 9.4

```
6500 REM cube/ lines (not stored)
6501 REM IN  : R(3,3)
6510 DATA 1,1,1, 1,1,-1, 1,-1,-1, 1,-1,1, -1,1,1, -1,1,-1, -1,-1,-1, -1,-1,1
6520 DATA 1,2, 2,3, 3,4, 4,1, 5,6, 6,7, 7,8, 8,5, 1,5, 2,6, 3,7, 4,8
6530 RESTORE cube
6539 REM input SETUP vertices of cube and move them into OBSERVED POSITION.
6540 FOR I = 1 TO 8
6550 READ XX,YY,ZZ
6560 LET X(I) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6570 LET Y(I) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6580 LET Z(I) = XX*R(3,1) + YY*R(3,2) + ZZ*R(3,3) + R(3,4)
6590 NEXT I
6599 REM input line information : draw lines by joining pairs of vertices.
6600 FOR I = 1 TO 12
6610 READ L1,L2
6620 LET XPT = X(L1): LET YPT = Y(L1): GO SUB moveto
6630 LET XPT = X(L2): LET YPT = Y(L2): GO SUB lineto
6640 NEXT I
6650 RETURN
```

We can have more than one cube in the scene. For example, if we rewrite 'scene3' as in listing 9.5, keeping all the other routines the same, we would get figure 9.2. Note that the X, Y and Z values of the previous cube are overwritten in the second call to 'cube'. Also, because we have the same ACTUAL to OBSERVED matrix for both cubes (they have different SETUP to ACTUAL matrices) we need to store *Q* so it can also be used for the second cube. Remember *Q* must premultiply the array P, which moves the second cube into the ACTUAL position. The data for figure 9.2 are HORIZ = 9, VERT = 6, (EX, EY, EZ) = (3, 2, 1) and (DX, DY, DZ) = (0, 0, 0).

Listing 9.5

```
6000 REM scene3/figure 9.2
6010 DIM X(8): DIM Y(8): DIM Z(8)
6020 DIM A(4,4): DIM B(4,4): DIM R(4,4): DIM Q(4,4)
6030 LET cube = 6500
6039 REM draw 1'st cube in OBSERVED position.
6040 GO SUB idR3: GO SUB look3
6050 GO SUB cube
6059 REM draw 2'nd cube in OBSERVED position.
6060 FOR I = 1 TO 4: FOR J = 1 TO 4
6070 LET Q(I,J) = R(I,J)
6080 NEXT J: NEXT I
6090 GO SUB idR3
6100 LET TX = 3: LET TY = 1.5: LET TZ = 2: GO SUB tran3: GO SUB mult3
6110 FOR I = 1 TO 4: FOR J = 1 TO 4
6120 LET A(I,J) = Q(I,J)
6130 NEXT J: NEXT I
6140 GO SUB mult3
6150 GO SUB cube
6160 RETURN
```

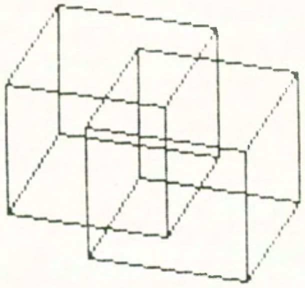


Figure 9.2

Exercise 9.1
Extend the routine 'cube' so that information about the size of a rectangular block is input, enabling the routine to construct a block of length LH, breadth BH and height HT: multiply the x-values of the SETUP cube by LH/2, the y-values by HT/2 and the z-values by BH/2.

Again it should be noted that the modular approach we have adopted may not be the most efficient method of drawing three-dimensional pictures. We chose this descriptive method in order to break down the complex situation into manageable pieces. Once the reader has mastered these concepts, he should *cannibalise* our programs for the sake of efficiency. However, to show the value of this modular approach we give another example, which illustrates just how quickly programs can be altered to draw new scenes and situations.

Example 9.2
We wish to view a fixed scene (for example, the one shown in figure 9.2) from a variety of observation points.

In this case it is better to store the vertex coordinates of the scene in the

ACTUAL position, rather than the OBSERVED position, and store the line information in array L. The 'scene3' routine (listing 9.6) must first set NOV and NOL to zero and then place the objects in their ACTUAL position using matrix $R = P$. The construction routine 'cube' (listing 9.7) must therefore be altered to update the data-base (but note the same routine could be used to store vertices in their OBSERVED position, which needs only a different $R = Q \times P$). Then for each different view point and direction the 'scene3' routine must clear the screen, set R to the identity matrix and call 'look3', and then call a special new 'drawit' routine (listing 9.8), which uses the matrix R (holding the values of Q, the ACTUAL to OBSERVED matrix) to put the points in the OBSERVED position and orthographically project them into arrays V and W (we cannot use X and Y because this would corrupt our ACTUAL data-base). Routine 'drawit', which was labelled in 'scene3', can then use the information in array L to draw the picture on the screen.

Listing 9.6

```
6000 REM scene3/figure 9.2 (variety of views)
6010 DIM X(16): DIM Y(16): DIM Z(16)
6020 DIM V(16): DIM W(16): DIM L(2,24)
6030 DIM A(4,4): DIM B(4,4): DIM R(4,4)
6040 LET cube = 6500: LET drawit = 7000
6050 LET NOV = 0: LET NOL = 0
6059 REM store 1'st cube in ACTUAL ( = SETUP ) position.
6060 GO SUB idR3
6070 GO SUB cube
6079 REM store 2'nd cube in ACTUAL position.
6080 LET TX = 3: LET TY = 1.5: LET TZ = 2: GO SUB tran3: GO SUB mult3
6090 GO SUB cube
6099 REM loop through different viewing positions.
6100 GO SUB idR3: GO SUB look3
6109 REM draw the two cubes in OBSERVED position.
6110 CLS: GO SUB drawit
6120 GO TO 6100
6130 RETURN
```

Listing 9.7

```
6500 REM cube/ vertices and lines (stored)
6501 REM IN  : NOV,NOL,X(NOV),Y(NOV),Z(NOV),L(2,NOL),R(4,4)
6502 REM OUT : NOV,NOL,X(NOV),Y(NOV),Z(NOV),L(2,NOL)
6510 DATA 1,1,1, 1,1,-1, 1,-1,-1, 1,-1,1, -1,1,1, -1,1,-1, -1,-1,-1, -1,-1,1
6520 DATA 1,2, 2,3, 3,4, 4,1, 5,6, 6,7, 7,8, 8,5, 1,5, 2,6, 3,7, 4,8
6530 RESTORE cube
6540 LET NV = NOV
6549 REM input and store vertices in position ( ACTUAL or OBSERVED ) using R.
6550 FOR I = 1 TO 8
6560 READ XX,YY,ZZ: LET NOV = NOV + 1
6570 LET X(NOV) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6580 LET Y(NOV) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6590 LET Z(NOV) = XX*R(3,1) + YY*R(3,2) + ZZ*R(3,3) + R(3,4)
6600 NEXT I
6609 REM input and store line information.
6610 FOR I = 1 TO 12
6620 READ L1,L2: LET NOL = NOL + 1
6630 LET L(1,NOL) = L1 + NV: LET L(2,NOL) = L2 + NV
6640 NEXT I
6650 RETURN
```

Listing 9.8

```
7000 REM drawit
7001 REM IN  : NOV,NOL,X(NOV),Y(NOV),Z(NOV),L(2,NOL),R(4,4)
7002 REM move vertices into OBSERVED position and draw object.
7010 FOR I = 1 TO NOV
7020 LET V(I) = X(I)*R(1,1) + Y(I)*R(1,2) + Z(I)*R(1,3) + R(1,4)
7030 LET W(I) = X(I)*R(2,1) + Y(I)*R(2,2) + Z(I)*R(2,3) + R(2,4)
7040 NEXT I
7050 FOR I = 1 TO NOL
7060 LET L1 = L(1,I): LET L2 = L(2,I)
7070 LET XPT = V(L1): LET YPT = W(L1): GO SUB moveto
7080 LET XPT = V(L2): LET YPT = W(L2): GO SUB lineto
7090 NEXT I
7100 RETURN
```

If the observer is travelling in a straight line and always looking in the same direction, we need not even calculate Q each time, but simply initially manipulate space so that the observer is looking along the z-axis, and then use the 'setorigin' routine to move the observer instead! After you have gained expertise in drawing three-dimensional projections, you should choose your construction and viewing method with care. You will rarely need to go through the complete method given in this chapter, there will always be short-cuts.

Exercise 9.2
Produce construction routines for a tetrahedron, pyramid, etc. For example,
(a) tetrahedron: vertices (1, 1, 1), (1, –1, –1), (–1, 1, –1) and (–1, –1, 1); lines 1 to 2, 1 to 3, 1 to 4, 2 to 3, 2 to 4 and 3 to 4.
(b) pyramid with square of side 1 and height HT: vertices (0, HT, 0), (1, 0, 1), (1, 0, –1), (–1, 0, –1) and (–1, 0, 1); lines 1 to 2, 1 to 3, 1 to 4, 1 to 5, 2 to 3, 3 to 4, 4 to 5 and 5 to 1.

Exercise 9.3
Set up a line drawing of any planar object in the x/y plane; for example, the outline of an alphabetic character or string of characters, and view them in various orientations in three-dimensional space. You can place such planar objects on the side of a cube. All you need do is extend the 'cube' routine above to include extra vertices and lines that define the symbols.

Thus far we have restricted our pictures to those of the simple cube. This is so that the methods we give are not obscured by the complexity in defining objects. Our programs will work for any object provided it fits within the limitations of our store (and time). For complex objects we merely extend the size of our arrays, although some objects will have properties that enable us to minimise store requirements. Consider the *jet* shown in figure 9.3 – it possesses two-fold symmetry, which can be used to our advantage. We assume that the plane of symmetry is the y/z plane, and so for every point (x, y, z) on the jet there is also a corresponding point $(-x, y, z)$. To draw figure 9.3 we use listings 9.1, 9.2 and

9.3, together with a construction routine 'jet' that generates all the vertices of the aeroplane with positive x-coordinates; thus information about only one-half of the jet is stored. To construct the complete aeroplane we need also a 'drawit' routine (listing 9.9), which draws one side of the jet, and then, by reversing the signs of all the x-values, draws the other.

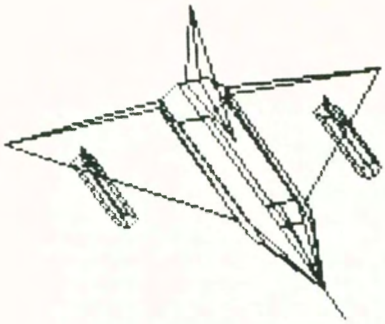


Figure 9.3

It is simple to construct these figures. Just plan your object in various sections on a piece of graph paper, number the important vertices and note which pairs of vertices are joined by lines. The coordinates values can be read directly from the grid on the paper. The data for figure 9.3 are HORIZ = 160, VERT = 120, (EX, EY, EZ) = (1, 2, 3) and (DX, DY, DZ) = (0, 0, 0).

Bodies of Revolution

This far in our construction of objects we have relied on DATA to input all the

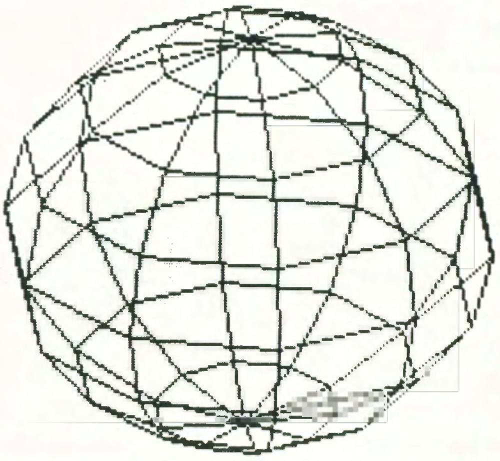


Figure 9.4

Listing 9.9

```
6000 REM scene3/jet
6010 DIM X(37): DIM Y(37): DIM Z(37)
6020 DIM L(2,46): DIM V(37): DIM W(37)
6030 DIM A(4,4): DIM B(4,4): DIM R(4,4)
6040 LET jet = 6500: LET drawit =7000
6050 GO SUB idR3: GO SUB look3
6060 GO SUB jet
6070 GO SUB drawit
6080 RETURN

6500 REM jet
6502 REM OUT : NOV,NOL,X(NOV),Y(NOV),Z(NOV),L(2,NOL)
6510 DATA 0,0,80,    0,0,64,    0,8,32,    4,8,32
6520 DATA 8,4,32,    8,0,32,    4,-4,32
6530 DATA 0,8,-32,   4,8,-32,   8,4,-32,   8,0,-32
6540 DATA 4,-4,-32,  0,-4,-32,  8,0,24,    48,0,-32
6550 DATA 8,2,-32,   0,8,0,     2,8,-32,   0,32,-32
6560 DATA 28,-4,-24, 30,-2,-24, 32,-2,-24, 34,-4,-24
6570 DATA 32,-6,-24, 30,-6,-24, 28,-4,8    30,-2,8
6580 DATA 32,-2,8,   34,-4,8,   32,-6,8,   30,-6,8
6590 DATA 31,0,-24,  31,-2,-24, 31,-2,-12, 31,0,-12
6600 DATA 0,6,40,    3,6,40
6610 DATA 1,2,    2,3,    2,4,    2,5,    2,6,    2,7,    3,4
6620 DATA 4,9,    5,10,   6,11,   7,12,   8,9,    9,10,   10,11,  11,12
6630 DATA 12,13,  14,15,  15,10,  15,16,  14,16,  17,18,  17,19,  18,19
6640 DATA 20,21,  21,22,  22,23,  23,24,  24,25,  25,20,  26,27,  27,28
6650 DATA 28,29,  29,30,  30,31,  31,26,  20,26,  21,27,  22,28,  23,29
6660 DATA 24,30,  25,31;  32,33,  33,34,  34,35,  35,32,  36,37
6663 REM SETUP vertex and line information for half the jet.
6700 LET NOV = 37: LET NOL = 46
6710 FOR I = 1 TO NOV: READ X(I),Y(I),Z(I): NEXT I
6720 FOR I = 1 TO NOL: READ L(1,I),L(2,I): NEXT I
6730 RETURN

7000 REM drawit/ two halves of the jet
7001 REM IN  : NOV,NOL,X(NOV),Y(NOV),Z(NOV),L(2,NOL),R(4,4)
7010 LET IS = 1
7019 REM loop through two halves of the jet.
7020 FOR J = 1 TO 2
7030 FOR I = 1 TO NOV
7040 LET XX = IS*X(I): LET YY = Y(I): LET ZZ = Z(I)
7049 REM put vertices in OBSERVED position.
7050 LET V(I) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
7060 LET W(I) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
7070 NEXT I
7080 FOR I = 1 TO NOL
7090 LET L1 = L(1,I): LET L2 = L(2,I)
7100 LET XPT = V(L1): LET YPT = W(L1): GO SUB moveto
7110 LET XPT = V(L2): LET YPT = W(L2): GO SUB lineto
7120 NEXT I
7130 LET IS = -1
7140 NEXT J
7150 RETURN
```

information about lines and vertices. We consider now a type of object where only a small amount of information is required for a quite complex object –this is a body of revolution, an example of which is shown in figure 9.4.

The method is simply to create a defining sequence of NUMV lines in the *x*/*y* plane through the origin; we call this the definition set. Then we revolve this set about the vertical (*y*-axis) NUMH–1 further times to create new vertical sets. The NUMV lines in the definition set are formed by joining the NUMV + 1 vertices (S(I), T(I), 0) (where $1 \leqslant I \leqslant$ NUMV + 1) in order. From this we generate NUMH different vertical sets, the Jth vertical set is the definition set rotated through an angle PHI + 2π(J – 1)/NUMH about the vertical *y*-axis, for some input value PHI (ϕ). As well as the set of NUMH × NUMV vertical lines we introduce horizontal lines also. We consider a single point (S(I), T(I), 0) at the end of a line segment in the definition set: as we rotate about the vertical axis it moves into NUMH positions (provided that the point is not on the axis of rotation)

$$(S(I) \times \cos(\theta + \phi), T(I), S(I) \times \sin(\theta + \phi))$$

where $\theta = 2\pi(J - 1)$ with $1 \leqslant J \leqslant$ NUMH.

These NUMH points are joined in order and the NUMHth position is joined back to the first, to give the Ith horizontal set. So there are (NUMH – *n*) × NUMV horizontal lines, where *n* is the number of vertices on the axis of rotation. Listing 9.10 is a construction routine 'rotbod', which draws the body of revolution when given NUMV, NUMH, PHI, the original set of vertices in T and S, and the positional matrix *R*. Listing 9.11 is the 'scene3' routine, which creates the scene of a spheroid in figure 9.4 by placing eight points from a semicircle into the definition set: HORIZ = 3.2, VERT = 2.2, PHI = $\pi/25$, NUMH = 10, NUMV = 8, viewed from (1, 2, 3) looking at (0, 0, 0). NUMV is assumed to be less than or equal to fifteen in listing 9.11.

Listing 9.10

```
6000 REM scene3/spheroid
6010 DIM X(32): DIM Y(32)
6020 DIM A(4,4): DIM B(4,4): DIM R(4,4)
6030 DIM S(16): DIM T(16)
6040 LET revbod = 6500
6050 INPUT "NUMBER OF HORIZONTAL LINES",NUMH
6060 INPUT "NUMBER OF VERTICAL LINES",NUMV
6070 INPUT "ANGLE PHI ";PHI
6080 LET THETA = PI/2: LET TD = PI/NUMV
6089 REM generate definition set.
6090 FOR I = 1 TO NUMV + 1
6100 LET S(I) = COS THETA: LET T(I) = SIN THETA
6110 LET THETA = THETA + TD
6120 NEXT I
6130 GO SUB idR3: GO SUB look3
6140 GO SUB revbod
6150 RETURN
```

Exercise 9.4

Experiment with this technique, any line sequence will do. Try an ellipsoid: this is essentially the same as the spheroid except that the definition set is produced

Listing 9.11

```
6500 REM revbod/ body of revolution
6501 REM IN  : PHI,NUMH,NUMV,S(NUMV+1),T(NUMV+1),R(4,4)
6509 REM create first vertical set.
6510 LET THETA = PHI: LET TD = PI*2/NUMH
6520 LET N1 = NUMV + 1: LET C = COS PHI: LET S =SIN PHI
6530 FOR I = 1 TO N1
6540 LET XX = S(I)*C: LET YY = T(I): LET ZZ = S(I)*S
6550 LET X(I) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6560 LET Y(I) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6570 NEXT I
6579 REM loop through second vertical set.
6580 FOR J = 1 TO NUMH
6590 LET THETA = THETA + TD: LET C = COS THETA: LET S = SIN THETA
6600 FOR I = 1 TO N1
6610 LET XX = S(I)*C: LET YY = T(I): LET ZZ = S(I)*S
6620 LET X(I + N1) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6630 LET Y(I + N1) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6640 NEXT I
6649 REM draw lines in first vertical set.
6650 LET XPT = X(1): LET YPT = Y(1): GO SUB moveto
6660 FOR I = 2 TO N1
6670 LET XPT = X(I): LET YPT = Y(I): GO SUB lineto
6680 NEXT I
6689 REM join corresponding horizontal vertices on the two vertical sets.
6690 FOR I = 1 TO N1
6700 LET XPT = X(I): LET YPT = Y(I): GO SUB moveto
6710 LET XPT = X(I + N1): LET YPT = Y(I + N1): GO SUB lineto
6719 REM copy second vertical set into first and repeat process.
6720 LET X(I) = XPT: LET Y(I) = YPT
6730 NEXT I
6740 NEXT J
6750 RETURN
```

from a semi-ellipse rather than a semicircle. There is no need to produce only convex bodies: lines can cut one another, cross to and fro over the *y*-axis and have *x*-values that move up and down.

This idea can be extended into a *body of rotation*. Now as the set of lines moves around the central axis, the *y*-values of the points do not stay fixed. They

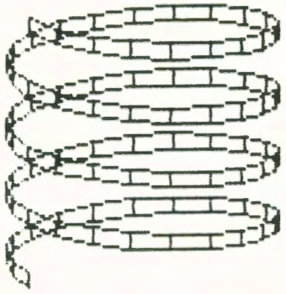


Figure 9.5

can move in a regular manner; that is, drop by the same amount with each rotation through 2π/NUMH. Now, of course, the lines can make more than one complete rotation about the axis, see figure 9.5. Write a program to implement a body of rotation.

Complete Programs

From now on we shall refer to listings 3.4 ('angle'), 8.1 ('mult3' and 'idR3'), 8.2 ('tran3'), 8.3 ('scale3'), 8.4 ('rot3'), 9.1 ('look3') and 9.2 ('main program') as 'lib3'.

I. 'lib1', 'lib3' and listings 9.3 ('scene3') and 9.4 ('cube'). Data required: HORIZ, VERT, (EX, EY, EZ) and (DX, DY, DZ). Try 6, 4, (1, 2, 3), (−1, 0, 1).

II. 'lib1', 'lib3' and listings 9.5 ('scene3') and 9.4 ('cube'). Data required: HORIZ, VERT, (EX, EY, EZ) and (DX, DY, DZ). Try 9, 6, (1, 2, 3), (−1, 0, 1). Make systematic changes to one of these input values and keep all the other parameters fixed.

III. 'lib1', 'lib3' and listings 9.6 ('scene3'), 9.7 ('cube') and 9.8 ('drawit'). Data required: HORIZ, VERT, and then repeated input of (EX, EY, EZ) and (DX, DY, DZ). Try 9, 6, then (1, 2, 3), (−1, 0, 1); (3, 2, 1), (0, 0, 1). Again make systematic changes to one of the input parameters.

IV. 'lib1', 'lib3' and listing 9.9 ('scene3', 'jet' and 'drawit'). Data required: HORIZ, VERT, and then repeated input of (EX, EY, EZ) and (DX, DY, DZ). Try 160, 120, then (1, 2, 31), (−1, 0, 30); (3, 2, 20), (0, 0, 21). Again make systematic changes to one of the input parameters.

V. 'lib1', 'lib3' and listings 9.10 ('scene3') and 9.11 ('revbod'). Data required: HORIZ, VERT, NUMH, NUMV ($\leqslant 15$), PHI, (EX, EY, EZ) and (DX, DY, DZ). Try 3.2, 2.2, 10, 10, 1, (1, 2, 3), (0, 0, 0); (3, 2, 1), (0, 0, 0).

10 Simple Hidden Line and Surface Algorithms

Having drawn a cube and other wire objects we soon become irritated by the lack of solidity in the figures. We would like to consider solid objects, in which case the facets at the front of the object will obviously restrict the view of the facets (and boundary lines) at the back. In order to draw pictures of such objects we have to introduce a hidden surface algorithm; or a hidden line algorithm if we wish to draw all, but only, the visible lines on the object. There are many, many such algorithms – some elementary for specially restricted situations, others very sophisticated for viewing general complicated scenes. The time limitations of microcomputers bar us from implementing the very complex algorithms. In the case of the Spectrum we have also the limitation that we cannot draw more than two colours in a character block, which therefore restricts us to line drawings. Nevertheless, by limiting the types and number of objects in the scenes, it is possible to get most acceptable pictures. In chapter 12 we discuss a relatively complex algorithm, but here we consider two special types of scene – we use the properties implicit in these special configurations to minimise the work needed to discover which surfaces and lines are hidden. Later in this chapter we shall give a simple method for drawing mathematically defined three-dimensional surfaces, but to start we consider an algorithm for drawing a single solid convex body in three-dimensional space.

For our work on hidden line and surface algorithms we choose to define a scene by storing the NOV vertices of objects in the scene (in the OBSERVED position) in arrays X, Y and Z as usual. However we shall now use facet information rather than line. The NOF facets are stored in an array F (we need also the array H for the number of edges on each polygonal facet), and to save space insist that no polygonal facet has more than six edges. Should we need more edges, then the facet must be broken down into a set of smaller polygons. In order to make the hidden surface algorithm easier we impose a restriction on the order of vertices within the array F. The vertices must be stored in the order in which they occur around the edge of the facet, and when viewed from the outside of an object they must be in an anti-clockwise orientation. Naturally from the inside the vertices taken in this same order would appear clockwise. We shall assume also that all lines are the junction of two facets. Individual lines not related to facets must be added as trivial two-sided facets.

The Orientation of a Three-dimensional Triangle

Once we have planned our object in terms of vertices and facets, how do we check that the facets are actually anti-clockwise? Simply write a program! The orientation of any convex polygon can be calculated from any three of its vertices taken in order, and so we need consider only an ordered triangle of vertices from the facet. We have already seen, in chapter 7, a method for calculating the orientation of a two-dimensional triangle. Our problem is solved if we can reduce the three-dimensional situation to two dimensions.

For simplicity we assume that all objects are SETUP about and containing the origin. We insist also that the infinite planes containing the facets on the surface of an object do not pass through the origin. Then we rotate space so that one of the vertices of the triangle in question lies on the negative *z*-axis (compare with routine 'look3', listing 9.1). Since we assume the origin is inside the object and the eye is outside, all we need do is project the transformed triangle back on to the *x*/*y* plane (that is, ignore the *z*-coordinates) and treat it like a two-dimensional triangle (in fact one of the three vertices will be (0,0)). Listing 10.1 is our solution of the problem.

Listing 10.1

```
100 REM  orientation of a 3-D triangle
110 DIM X(3): DIM Y(3): DIM Z(3)
120 DIM A(4,4): DIM B(4,4): DIM R(4,4)
130 LET rot3 = 8600: LET angle = 8800: LET mult3 = 9100: LET idR3 = 9300
140 LET J$ = "TYPE IN COORDINATES OF TRIANGLE "
150 FOR I = 1 TO 3
160 LET I$ = "VERTEX(" + STR$ I + ")=("
170 INPUT (J$ + I$);X(I);",";Y(I);",";Z(I);")"
180 PRINT AT 2+2*I,0;I$;X(I);",";Y(I);",";Z(I);")"
190 NEXT I
199 REM find matrix R that will put (X(1),Y(1),Z(1)) on negative z-axis.
200 GO SUB idR3
210 LET AX = X(1): LET AY = Y(1): GO SUB angle
220 LET AXIS = 3: LET THETA = -THETA: GO SUB rot3: GO SUB mult3
230 LET AX = Z(1): LET AY = SQR (X(1)*X(1) + Y(1)*Y(1)): GO SUB angle
240 LET AXIS = 2: LET THETA = PI - THETA: GO SUB rot3: GO SUB mult3
249 REM transform triangle so that all vertices lie in an x/y plane.
250 FOR I = 1 TO 3
260 LET XX = X(I): LET YY = Y(I): LET ZZ = Z(I)
270 LET X(I) = XX*R(1,1) + YY*R(1,2)+ ZZ*R(1,3) + R(1,4)
280 LET Y(I) = XX*R(2,1) + YY*R(2,2)+ ZZ*R(2,3) + R(2,4)
290 NEXT I
299 REM check if the now 2-D triangle is clockwise or anti-clockwise.
300 PRINT AT 11,0;"IF THE EYE AND THE ORIGIN ARE ON"
310 PRINT AT 13,0;"OPPOSITE SIDES OF THE FACET THEN"
320 PRINT AT 15,0;"THE TRIANGLE IS ";
330 LET DX1 = X(2) - X(1): LET DY1 = Y(2) - Y(1)
340 LET DX2 = X(3) - X(2): LET DY2 = Y(3) - Y(2)
350 IF DX1*DY2 - DX2*DY1 > 0 THEN PRINT "ANTI-";
360 PRINT "CLOCKWISE."
370 STOP
```

Exercise 10.1
Rewrite the wire-figure routines of the last chapter using the assumption that the data are given as vertices and anti-clockwise polygonal facets, and not as lines. Check your facet data with the above program. The line information is still there of course, implicit in the facet data – they are the edges of the facet considered as pairs of vertices. Within this information each line occurs twice, once on each of two neighbouring facets. We do not want to waste time drawing lines twice! Because of the anti-clockwise manner of constructing the figures we note that if a line joins vertex I to vertex J on one facet then the equivalent line on the neighbouring facet joins vertex J to I. So for wire figures stored as facets we shall draw lines from vertex I to vertex J if and only if $I < J$.

A Hidden Surface Algorithm for a Single Closed Convex Body

A finite *convex body* is one in which any line segment joining two points inside the body lies totally within the body: a direct extension of the definition in two-dimensional space. It is automatically closed, and thus it is impossible to get inside the body without crossing through its surface. We orthographically project all the vertices of the object on to the view plane, noting that a projection of a convex polygon with n sides in three-dimensional space is an n-sided convex polygon (or degenerates to a line) in the view plane. Taking the projected vertices of any facet in the same order as the original, we find that either the new two-dimensional polygon is in anti-clockwise orientation, in which case we are looking at the outside of the facet, or the new vertices are clockwise and we are looking at the underside. Since the object is closed we are able to see only the outside of facets; the view of their underside is blocked by the bulk of the object. Therefore we need draw only the anti-clockwise polygonal facets – a very simple algorithm, which can be implemented in either construction or 'drawit' routines.

For example, an adjusted construction routine 'cube' for eliminating the hidden lines from an orthographic picture of a cube is given as listing 10.2. Here we do not store the facets, but instead READ the information from DATA and draw the visible facets immediately. This program was used to produce figure 10.1, a hidden line version of figure 9.1d. We use all the routines from the last chapter that were used to draw figure 9.1d, except of course for the construction routine that sets up the data as vertices and facets, and draws the object (this replaces listing 9.4 in the program for drawing figure 9.1d). Naturally we use the same data that were used for figure 9.1d.

Exercise 10.2
Change listing 10.2, so that it can draw a rectangular block of length LH, breadth BH and height HT, where LH, BH and HT are input parameters to the routine. Then draw a hidden line picture of it. Draw hidden line pictures of tetrahedra, pyramids, octahedra, etc. Add extra parameters to distort these figures so that they are no longer regular, but are still convex.

Listing 10.2

```
6500 REM cube/facets (not stored) hidden line elimination
6501 REM IN  : R(4,4)
6510 DATA 1,1,1, 1,1,-1, 1,-1,-1, 1,-1,1, -1,1,1, -1,1,-1, -1,-1,-1, -1,-1,1
6520 DATA 1,2,3,4, 5,8,7,6, 1,5,6,2, 2,6,7,3, 3,7,8,4, 4,8,5,1
6530 RESTORE cube
6539 REM place vertices in OBSERVED position.
6540 FOR I = 1 TO 8
6550 READ XX,YY,ZZ
6560 LET X(I) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6570 LET Y(I) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6580 NEXT I
6590 FOR I = 1 TO 6
6599 REM READ facet information and draw it if oriented anti-clockwise.
6600 READ F1,F2,F3,F4
6610 LET DX1 = X(F2)-X(F1): LET DY1 = Y(F2)-Y(F1)
6620 LET DX2 = X(F3)-X(F2): LET DY2 = Y(F3)-Y(F2)
6630 IF DX1*DY2 - DX2*DY1 < 0 THEN GO TO 6690
6640 LET XPT = X(F1): LET YPT = Y(F1): GO SUB moveto
6650 LET XPT = X(F2): LET YPT = Y(F2): GO SUB lineto
6660 LET XPT = X(F3): LET YPT = Y(F3): GO SUB lineto
6670 LET XPT = X(F4): LET YPT = Y(F4): GO SUB lineto
6680 LET XPT = X(F1): LET YPT = Y(F1): GO SUB lineto
6690 NEXT I
6700 RETURN
```

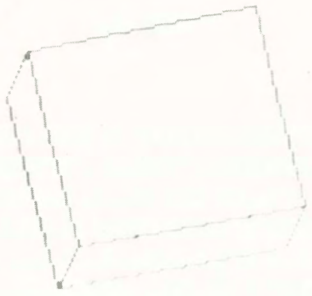


Figure 10.1

Bodies of Revolution

We can use this anti-clockwise versus clockwise method to produce hidden surface pictures of the bodies of revolution that were defined in chapter 9. As we go through the NUMH revolutions we generate NUMV facets with each move. Provided these quadrilateral (or perhaps degenerate triangular) facets are carefully constructed in an anti-clockwise orientation, then we can use the same algorithm. Listing 10.3 is just such a routine; it produces figure 10.2, a hidden surface version of figure 9.4 (and uses the same input data). Again, because of the modular design of our programs, all the routines needed to draw figure 10.2, except 'revbod', are the same as those given in chapter 9. Now, however, we must deal solely with convex bodies of revolution.

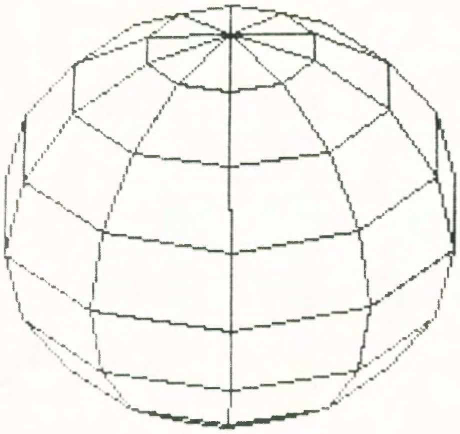


Figure 10.2

As the routine rotates the definition set of lines about the vertical axis, it stores the vertices of two consecutive vertical sets of lines. These form the vertical edges of one *slice* of facets. The vertices on these facets are immediately transformed by *R* (the SETUP to OBSERVED matrix) and stored in arrays X and Y. In such a configuration of pairs of vertical lines the first set of vertices have indices from 1 to NUMV + 1 (= N1), and the second from N1 + 1 to 2*N1. The Ith facet is bounded by four lines, two vertical joining vertex I to I + 1, and I + N1 to I + N1 + 1, and two horizontal joining I to I + N1, and I + 1 to I + N1 + 1. Adjustments must be made if one of the original vertices is on the axis of rotation, in which case the quadrilateral degenerates to a triangle. The order of vertices in each facet is carefully chosen so that they are in anti-clockwise orientation when viewed from outside the object. This allows us to use our simple algorithm to draw the object with the hidden lines suppressed. This technique was used also to draw figure I.1 in the introduction.

Exercise 10.3
Experiment with this technique. Any initial set of lines will do, provided that it starts and ends on the vertical axis and the polygon thus formed in the *x*/*y* plane is convex.

Listing 10.3

```
6500 REM revbod/ convex body of revolution (hidden line elimination)
6501 REM IN  : PHI,NUMH,NUMV,S(NUMV+1),T(NUMV+1),R(4,4)
6510 LET THETA = PHI: LET TD = PI*2/NUMH
6520 LET N1 = NUMV + 1: LET C = COS PHI: LET S = SIN PHI
6529 REM create first vertical set.
6530 FOR I = 1 TO N1
6540 LET XX = S(I)*C: LET YY = T(I): LET ZZ = S(I)*S
6550 LET X(I) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6560 LET Y(I) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6570 NEXT I
6579 REM loop through second vertical set.
6580 FOR J = 1 TO NUMH
6590 LET THETA = THETA + TD: LET C = COS THETA: LET S = SIN THETA
6600 FOR I = 1 TO N1
6610 LET XX = S(I)*C: LET YY = T(I): LET ZZ = S(I)*S
6620 LET X(I + N1) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6630 LET Y(I + N1) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6640 NEXT I
6649 REM take anticlockwise triangle from each facet between the two sets.
         If it keeps its orientation on projection then it is visible.
6650 FOR I = 1 TO NUMV
6660 LET F1 = I: LET F2 = I + 1: LET F3 = F2 + N1
6670 IF I = NUMV THEN LET F3 = F3 - 1
6680 LET DX1 = X(F2) - X(F1): LET DY1 = Y(F2) - Y(F1)
6690 LET DX2 = X(F3) - X(F2): LET DY2 = Y(F3) - Y(F2)
6700 IF DX1*DY2 - DX2*DY1 < 0 THEN GO TO 6770
6710 LET F3 = F2 + N1: LET F4 = F3 - 1
6720 LET XPT = X(F1): LET YPT = Y(F1): GO SUB moveto
6730 LET XPT = X(F2): LET YPT = Y(F2): GO SUB lineto
6740 LET XPT = X(F3): LET YPT = Y(F3): GO SUB lineto
6750 LET XPT = X(F4): LET YPT = Y(F4): GO SUB lineto
6760 LET XPT = X(F1): LET YPT = Y(F1): GO SUB lineto
6770 NEXT I
6779 REM copy second set into first and repeat process.
6780 FOR I = 1 TO N1
6790 LET X(I) = X(I + N1): LET Y(I) = Y(I + N1)
6800 NEXT I
6810 NEXT J
6820 RETURN
```

Drawing a Special Three-dimensional Surface

The call for pictures of convex solids is limited, so we now look at one type of non-convex figure that can be drawn using information about its special form. We consider the construction of a restricted type of three-dimensional surface in which the y-coordinate of each point on the surface is given by a single-valued function 'f' of the x-coordinate and z-coordinate of that point; 'f' will be included as a routine in the program – one such example is given in listing 10.4, the function $y = 4 \times \text{SIN}(\text{XZ})/\text{XZ}$ where $\text{XZ} = \sqrt{(x^2 + z^2)}$ shown in figure 10.3. The data required were HORIZ = 32, VERT = 22, (EX, EY, EZ) = (3, 2, 1), (DX, DY, DZ) = (0, 0, 0), NX = NZ = 16, XD = ZD = –10 and XT = ZT = 10.

Listing 10.4

```
6000 REM scene3
6010 DIM A(4,4): DIM B(4,4): DIM R(4,4)
6020 LET surface = 6500: LET drawlin = 7000: LET f = 7200
6030 GO SUB idR3: GO SUB look3
6040 GO SUB surface
6050 RETURN

6500 REM surface
6501 REM IN  : R(4,4)
6510 DIM D(256)
6520 INPUT "NX,XMIN,XMAX ";NX;",";XMIN;",";XMAX
6530 INPUT "NZ,ZMIN,ZMAX ";NZ;",";ZMIN;",";ZMAX
6540 LET XDIF = (XMAX - XMIN)/NX
6550 LET ZDIF = (ZMAX - ZMIN)/NZ
6559 REM draw zero'th set of fixed-x lines.
6560 LET XX = XMAX: LET ZZ = ZMIN: GO SUB f
6570 LET XA = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6580 LET YA = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6590 FOR J = 1 TO NZ
6600 LET ZZ = ZZ + ZDIF: GO SUB f
6610 LET XB = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6620 LET YB = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6630 GO SUB drawlin
6640 LET XA = XB: LET YA = YB
6650 NEXT J
6659 REM draw zero'th set of fixed-z lines.
6660 FOR J = 1 TO NX
6670 LET XX = XX-XDIF: GO SUB f
6680 LET XB = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6690 LET YB = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6700 GO SUB drawlin
6710 LET XA = XB: LET YA = YB
6720 NEXT J
6729 REM move x values back in NX steps.
6730 LET XS = XMAX
6740 FOR I = 1 TO NX
6748 REM draw visible parts of one from each of the fixed-z lines:
6749     the x values vary from (I-1)st x-line to the I'th.
6750 LET ZS = ZMAX
6760 FOR J = 1 TO NZ
6770 LET ZS = ZS - ZDIF
6780 LET XX = XS: LET ZZ = ZS: GO SUB f
6790 LET XA = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6800 LET YA = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6810 LET XX = XS-XDIF: GO SUB f
6820 LET XB = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6830 LET YB = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6840 GO SUB drawlin
6850 NEXT J
6859 REM draw visible parts of the fixed-x lines.
6860 LET ZZ = ZMIN: GO SUB f
6870 LET XA = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6880 LET YA = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6890 FOR J = 1 TO NZ
6900 LET ZZ = ZZ + ZDIF: GO SUB f
6910 LET XB = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6920 LET YB = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6930 GO SUB drawlin
6940 LET XA = XB: LET YA = YB
6950 NEXT J
6960 LET XS = XS-XDIF
6970 NEXT I
6980 RETURN
```

```
7000 REM drawlin/ draw line between points
7001 REM IN  : XA,YA,XB,YB,D(256)
7008 REM visible parts of line between (XA,YA) and (XB,YB).
7009 REM IA and IB are the x-pixel positions of the two points.
7010 LET IA = FN X(XA): LET IB = FN X(XB)
7020 LET Y = YA: LET YD = 0
7030 IF IA <> IB THEN GO TO 7080
7039 REM if IA=IB then only draw line if second point is above first.
7040 PLOT IA,D(IA)
7050 LET IY = FN Y(YB)
7060 IF IY > D(IA) THEN DRAW 0,IY - D(IA): LET D(IA) = IY
7070 RETURN
7079 REM move in pixel columns from left to right.
7080 LET YD = (YB - YA)/(IB - IA)
7090 FOR K = IA TO IB
7100 LET IY = FN Y(Y)
7109 REM if y-pixel is greater than D, then reset D-value.
7110 IF D(K) < IY THEN LET D(K) = IY
7120 LET Y = Y + YD
7130 NEXT K
7139 REM join all the points (I,D(I)) where IA<=I<=IB.
7140 PLOT IA,D(IA)
7150 FOR K = IA + 1 TO IB
7160 DRAW 1,D(K) - D(K - 1)
7170 NEXT K
7180 RETURN

7200 REM f/ function to be drawn
7201 REM IN  : XX,ZZ
7202 REM OUT : YY
7210 LET YY = 4: LET XZ = SQR (XX*XX + ZZ*ZZ)
7220 IF XZ > 0.000001 THEN LET YY = 4*SIN (XZ)/XZ
7230 RETURN
```

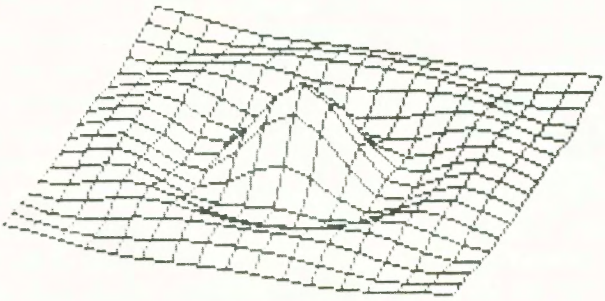


Figure 10.3

Since it is impossible to draw every point on the surface, we have to approximate by considering a subset of these surface points. We choose those points with x/z coordinate on a grid; in other words when orthographically viewed directly from above (thus ignoring the y-values), the points form a rectangular grid. This grid is composed of NX by NZ rectangles in the x/z plane. The x-coordinates of the vertices are equi-spaced and vary between XB and XT (XB < XT) and the equi-spaced z-values vary between ZB and ZT (ZB < ZT). There are thus

$(NX + 1) \times (NZ + 1)$ vertices (X, Z) in the grid that can be identified by the pair of integers (i, j)

$$X = XT + i(XB - XT)/NX \quad 0 \leqslant i \leqslant NX$$

$$Z = ZT + j(ZB - ZT)/NZ \quad 0 \leqslant j \leqslant NZ$$

The equivalent point on the surface is (X, Y, Z) where $Y = f(X, Z)$. Every one of the $(NX + 1) \times (NZ + 1)$ points generated in this way is joined to its four immediate neighbours along the grid (that is, those with an equal x-value or z-value), unless it lies on the edge, in which case it is joined to three, or in the case of corners to two, neighbours. The grid lines can be considered as NX + 1 sets of different fixed-x line segments and NZ + 1 sets of fixed-z line segments. For example the Ith fixed-x set of lines $(0 \leqslant I \leqslant NX)$ consists of NZ lines joining pairs of points equivalent to the grid points (I, J) to (I, J + 1), where $0 \leqslant J \leqslant NZ - 1$.

The surface can undulate, so not all the lines need be visible from a given view point. We devise a very simple method to eliminate the hidden lines by working from the front of the picture to the back.

To simplify the algorithm we assume that the eye is always in the positive quadrant (that is, $EX > 0$ and $EZ > 0$), and that the eye is always looking at the origin $(DX = DY = DZ = 0)$. If the function is non-symmetrical and we wish to view it from another quadrant, then we simply change the sign of x and/or z in the function. We can then transform the surface into the OBSERVED position.

In this position we first orthographically project the front edge, the two grid line with $x = XT$ and $z = ZT$, on to the view plane. That is, the zeroth set of fixed-x lines and the zeroth set of fixed z-lines. We now work from the front to the back in NX steps. In the Ith step $(1 \leqslant I \leqslant NX)$, we draw first the visible parts of one line segment from each of the NZ fixed-z sets; the Jth such line $(1 \leqslant J \leqslant NZ)$ joins the points equivalent to the grid points (I − 1, J) to (I, J). That is, we join the corresponding points on the (I − 1)th fixed-x line to the Ith, starting at the zeroth fixed-z line and working backwards. Then we draw the visible parts of the Ith set of fixed-x line segments. This is all programmed in routine 'surface'.

As yet we have not explained how to draw the visible parts of the lines: routine 'draw1in'. We define an array D of 256 values, one for each column of pixels across the screen. When we first draw the zeroth sets of fixed-x and fixed-z lines, we put the pixel values of the points on these lines in the array D. We calculate the row/column values for every pixel on a given line segment and draw it if and only if the row value for a column is greater than the value stored in D. Whenever a pixel is added to the diagram then the value of array D is altered accordingly. This method furnishes us with a hidden line elimination algorithm for mathematical surfaces because of the order in which we consider the lines. If we move out of the positive quadrant, or let the scale of the figure exceed the size of the graphics area, then we shall incur errors.

Exercise 10.4
Change the functions 'f' used by this program. For example, use $f = 4 \times \mathrm{SIN}(t)$ where $t = \sqrt{(x^2 + z^2)}$.

Exercise 10.5
The above program does not draw the underside of the surface if it shows below the zeroth fixed-x/z lines. Alter listing 10.4 to correct this shortcoming. Define another array E(256) that initially holds information on the nearest edges, and whenever pixels on the line segments go below the values stored in E then the pixel is drawn and the value of E altered. Can we use the same order for drawing the lines?

Complete Programs

I. 'lib3' and listing 10.1. Data required: the vertex coordinates of a triangle (X(I), Y(I), Z(I)), where $1 \leqslant I \leqslant 3$. Try (1, 0, 1), (1, 1, 0) and (0, 1, 1): also the same vertices in a different order (1, 1, 0), (1, 0, 1) and (0, 1, 1).

II. 'lib1', 'lib3' and listings 9.3 ('scene3') and 10.2 ('cube'). Data required: HORIZ, VERT, (EX, EY, EZ), (DX, DY, DZ). Try 9, 6, (1, 2, 3), (0, 0, –1).

III. 'lib1', 'lib3' and listings 9.10 ('scene3') and 10.3 ('revbod'). Data required: HORIZ, VERT, NUMH, NUMV ($\leqslant 15$), PHI, (EX, EY, EZ), (DX, DY, DZ). Try 3.2, 2.2, 10, 10, 1, (1, 2, 3), (0, 0, –1).

IV. 'lib1', 'lib3' and listing 10.4 ('scene3', 'surface', 'drawlin' and 'f'). Data required: HORIZ, VERT, (EX, EY, EZ), (DX, DY, DZ), NX, XB, XT, NZ, ZB, ZT. Try 30, 20, (1, 2, 3), (0, 0, 0), 16, –8, 8, 16, –8, 8.

11 Perspective Projections

We have seen that the orthographic projection has the property that parallel lines in three-dimensional space are projected into parallel lines on the view plane. Although very useful, such views do look odd! Our brains are used to the perspective phenomenon of three-dimensional space, and so they attempt to interpret orthographic figures as if they are perspective views. For example, the cubes of figures 9.1 and 10.1 look distorted.

So it is essential to produce a projection that displays perspective phenomena (that is, parallel lines should meet on the horizon); an object should appear smaller as it moves away from the observer. The *drawing-board* methods devised by artists over the centuries are of no value to us. Three-dimensional coordinate geometry and the concept of ACTUAL to OBSERVED positions, however furnish us with a relatively straightforward technique.

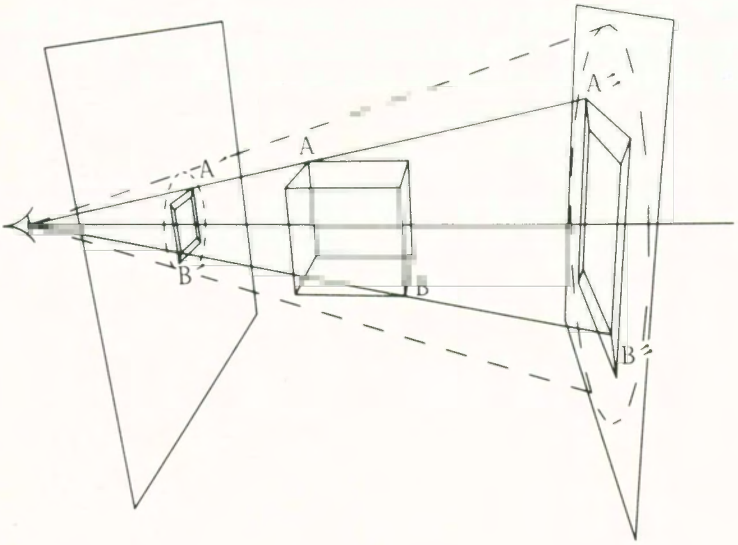


Figure 11.1

What is Perspective Vision?

To produce a perspective view we introduce a very simple definition of what we mean by vision. We imagine every visible point in space sending out a ray that enters the eye. Naturally the eye cannot see all of space, it is limited to a cone of rays that fall on the retina, the so-called *cone of vision*, which is outlined by the dashed lines of figure 11.1. The axis of this cone is called the *straight-ahead ray.* We imagine that space has been transformed into the OBSERVED position with the eye at the origin and the straight-ahead ray identified with the positive z-axis.

We place the view plane (which we call the *perspective plane* in this special case) perpendicular to the axis of the cone of vision at a distance d from the eye. In order to form the perspective projection we mark the points of intersection of each ray with this plane. Since there are an infinity of such rays this appears to be an impossible task. Actually the problem is not that great because we need consider only those rays that emanate from the important points in the scene; that is, the vertices at the ends of line segments or the corners of polygonal facets. The final view is formed by relating the projected points on the perspective plane in exactly the same way as they are related in three-dimensional space, and then identifying the view plane with the graphics screen.

Figure 11.1 shows a cube observed by an eye and projected on to two planes, and the whole scene is drawn in perspective! two example rays are shown: the first from the eye to A, one of the near corners of the cube (relative to the eye), and the second to B, one of the far corners of the cube. The perspective projections of these points on to the near plane are A' and B', and on to the far plane A'' and B''. Note that the projections will have the same shape and orientation, but they will be of different sizes.

Calculation of the Perspective Projection of a Point

We let the perspective plane be a distance d from the eye (variable PPD in later programs). Consider a point $P \equiv (x, y, z)$ in space that sends a ray into the eye. We must calculate the point where this line cuts the view plane (the $z = d$ plane); suppose it is the point $P' \equiv (x', y', d)$. Let us first consider the value of y' by referring to figure 11.2. By similar triangles we see that $y'/d = y/z$, that is $y' = y \times d/z$. Similarly $x' = x \times d/z$. Hence $P' \equiv (x \times d/z, y \times d/z, d)$. Since the view plane is identified with the x/y coordinate system of the graphics screen, we can ignore the $z = d$ coordinate.

Example 11.1

Calculate the perspective projection of a cube with eight vertices $(0, 0, 4) + (\pm 1, \pm 1, \pm 1)$ on the perspective plane $z = 4$, where the eye is origin and the straight-ahead ray is the positive z-axis.

The space is defined so that the scene is in the OBSERVED position. We can

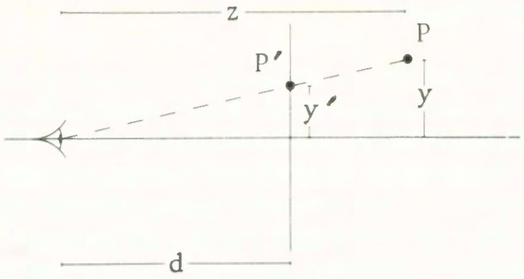


Figure 11.2

calculate the projections of the eight vertices using the above method. For example, (1, 1, 3) is projected to (1 × 4/3, 1 × 4/3, 4) = (4/3, 4/3, 4) → (4/3, 4/3) on the screen. So we get the eight projections

(1, 1, 3) → (4/3, 4/3)	(1, −1, 3) → (4/3, −4/3)
(−1, 1, 3) → (−4/3, 4/3)	(−1, −1, 3) → (−4/3, −4/3)
(1, 1, 5) → (4/5, 4/5)	(1, −1, 5) → (4/5, −4/5)
(−1, 1, 5) → (−4/5, 4/5)	(−1, −1, 5) → (−4/5, −4/5)

and the resulting diagram is shown in figure 11.3a.

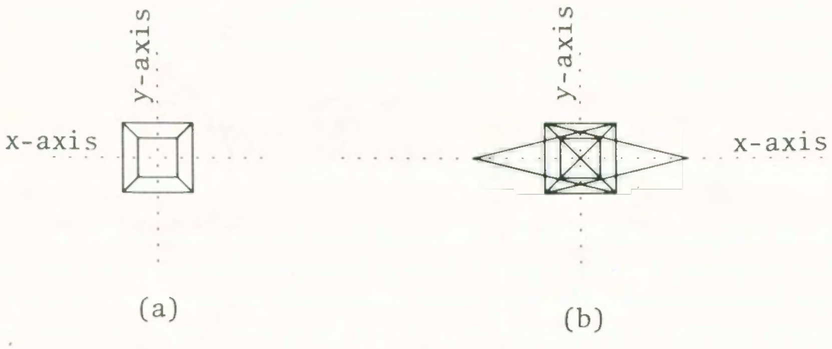


Figure 11.3

Properties of the Perspective Transformation

(1) The perspective transformation of a straight line (Γ_3 say) is a straight line (Γ_2 say). This is obvious because the origin (the eye) and the line Γ_3 form a plane (Ω say) in three-dimensional space and all the rays emanating from points on Γ_3 lie in this plane. (If the line enters the eye, Ω degenerates into a line). Naturally Ω cuts the perspective plane in a line Γ_2 (or degenerates to a point)

and so the perspective projection of a point on the original line Γ_3 now lies on the new line Γ_2. It is important to realise that a line does not become curved on perspective projection.

(2) The perspective transformation of a facet (a closed sequence of coplanar line segments) is a facet in the perspective plane. If the facet is an area bounded by n coplanar line segments, then the transform of this facet is naturally an area in the $z = d$ plane bounded by the transforms of the n line segments. Again note that no curves are introduced in this projection: if they were, then the task of producing perspective pictures would be far more complicated.

(3) The projection of a convex facet is also convex. Suppose facet F_1 is projected on to facet F_2. Since the projection of a closed facet is also closed and lines go into lines, then points inside F_1 are projected into points inside F_2. Suppose F_2 is not convex: then there exist two points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ inside F_2 such that the line joining them goes outside this facet. Hence there is at least one point $\boldsymbol{p}$ on the line outside F_2. If $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ are projections of points $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ from F_1, then $\boldsymbol{p}$ is the projection of some point $\boldsymbol{q}$ on the line joining $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$. Since the F_1 is convex then $\boldsymbol{q}$ must be inside F_1 and thus $\boldsymbol{p}$ must be inside F_2: a contradiction, and our proposition is proved.

(4) All infinitely long parallel lines appear to meet at one point, their so-called *vanishing point*. If we take a general line (with base vector $\boldsymbol{p}$) from a set of parallel lines with direction vector $\boldsymbol{h}$

$$\boldsymbol{p} + \mu\boldsymbol{h} \equiv (x_p, y_p, z_p) + \mu(x_h, y_h, z_h)$$

where $z_h > 0$; then the perspective transform of a general point on this line is

$$\left(\frac{(x_p + \mu x_h) \times d}{(z_p + \mu z_h)}, \frac{(y_p + \mu y_h) \times d}{(z_p + \mu z_h)} \right)$$

which can be rewritten as

$$\left(\frac{(x_h + x_p/\mu) \times d}{(z_h + z_p/\mu)}, \frac{(y_h + y_p/\mu) \times d}{(z_h + z_p/\mu)} \right)$$

As we move along the line towards large z-coordinates (that is, as $\mu \to \infty$), then the line moves towards its vanishing point, which is therefore given by $(d \times x_h/z_h, d \times y_h/z_h)$. This vanishing point is independent of $\boldsymbol{p}$, the base point of the line, and hence all lines parallel to the direction $\boldsymbol{h}$ have the same vanishing point. Of course the case $z_h < 0$ is ignored because the line would disappear outside the cone of vision as $\mu \to \infty$.

(5) The vanishing points of all lines in parallel planes are collinear. Suppose that the set of parallel planes has a common normal direction $\boldsymbol{n} \equiv (x_n, y_n, z_n)$. If a general line in one of these planes has direction $\boldsymbol{h} \equiv (x_h, y_h, z_h)$, then $\boldsymbol{h}$ is per-

pendicular to ***n*** (all lines in these planes are perpendicular to the normal to the plane ***n***). Thus ***n*** · ***h*** = 0, which in coordinate form is

$$x_n \times x_h + y_n \times y_h + z_n \times z_h = 0$$

dividing by z_h gives

$$x_n \times x_h/z_h + y_n \times y_h/z_h + z_n = 0$$

and so the vanishing point $(d \times x_h/z_h, d \times y_h/z_h)$ lies on the straight line

$$x_n \times x + y_n \times y + d \times z_n = 0$$

and the statement is proved.

Example 11.2
Find the vanishing points of the edges of the cube in example 11.1, and of the diagonals of its top and bottom planes.

We divide the twelve edges of the cube into three sets of four edges, each set being parallel to the x-axis, y-axis and z-axis respectively and so with directional vectors $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$. The first two sets have zero z-values, and so their extended edges disappear outside the cone of vision and are ignored, whereas the third direction has vanishing point $(4 \times 0/1, 4 \times 0/1) \equiv (0, 0)$ on the view plane. On the top and bottom faces the diagonals have directions $(1, 0, 1)$, the major diagonal, and $(-1, 0, 1)$, the minor diagonal. The major diagonal on the top plane is $(-1, 1, 3) + \mu(1, 0, 1)$, and so the vanishing point is $(4 \times 1/1, 4 \times 0/1) \equiv (4, 0)$. The minor diagonal on the top plane is $(1, 1, 3) + \mu(-1, 0, 1)$ and the vanishing point is $(4 \times -1/1, 4 \times 0/1) \equiv (-4, 0)$. By similiar calculations we find that the vanishing points of the major and minor diagonals on the lower face are also $(4, 0)$ and $(-4, 0)$ respectively. The relevant edges are extended to their vanishing points in figure 11.3b. Note that all the lines mentioned lie in the two parallel planes (the top and bottom faces of the cube) and so the vanishing points should be collinear: since $(4, 0)$, $(0, 0)$ and $(-4, 0)$ all lie on the x-axis, this is obviously true. It can be shown similarly that the vanishing points of the diagonals of the side faces lie on a vertical line through the origin.

Exercise 11.1
Draw a perspective view of a tetrahedron with vertices $(1, 1, 5)$, $(1, -1, 3)$, $(-1, 1, 3)$ and $(-1, -1, 5)$. Find the vanishing points (inside the cone of vision) of lines that join pairs of mid-points of edges of the tetrahedron.

Programming the Perspective Transformation

The main program for drawing a perspective view of any scene is the same as that for the orthographic view, namely listing 9.2. Again the overall scene is created by a call to a routine 'scene3' similar to those discussed in chapter 9. We shall often need to calculate explicitly the ACTUAL to OBSERVED matrix, so that the eye is in the OBSERVED position at the origin looking along the positive z-axis. This is achieved by routine 'look3', given in chapter 9 (listing 9.1). Calls are made to construction routines, each having a matrix R as parameter. Finally the figure must be drawn, inside the construction routines or in a 'drawit' routine.

The only difference between the program that draws a perspective view and the program of chapter 9 (orthographic view) is in the calculation of the co-ordinates of the projected image on the view plane. Unlike the orthographic, in the perspective projection the coordinates on the view plane cannot be identified with the x-value and y-value of the point in the OBSERVED position. We need to store the perspective transformation of the vertices in the arrays V and W: the I$^{\text{th}}$vertex (X(I), Y(I), Z(I)) in the OBSERVED position is projected to (V(I), W(I)). The values in arrays V and W are given by

V(I) = X(I)*PPD/Z(I) and W(I) = Y(I)*PPD/Z(I) for I = 1, 2, . . ., NOV

where the value of PPD is set to 3*VERT; the reason for this equation is given in the next section. The calculation of V and W can be made in the construction routine in the 'scene3' or 'drawit' routines; this simply depends on the scene being considered.

Example 11.3

We draw a fixed scene (the two cubes described in example 9.2) in perspective from a variety of observation points, setting HORIZ = 9 and VERT = 6. The necessary 'scene3' routine is given in listing 11.1 below; note that this calculates PPD (compare with listing 9.6). This listing places the group of cubes in their ACTUAL position using the 'cube' routine of listing 9.7, and then loops through a number of different OBSERVER positions. For each time through the loop we call 'look3', which requires (EX, EY, EZ) and (DX, DY, DZ) to calculate the ACTUAL to OBSERVER matrix. Then the perspective 'drawit' routine (listing 11.2) is called. This uses the matrix to transform the vertices from their (stored) ACTUAL position to the OBSERVER position, and places the projected vertex coordinates in arrays V and W, according to the above equations. The routine can then finally draw the edges of the cubes in perspective.

Figure 11.4 was drawn using (EX, EY, EZ) ≡ (15, 10, 5) and (DX, DY, DZ) ≡ (0, 0, 0). Compare this with the orthographic view of the same scene given in figure 9.2.

Listing 11.1

```
6000 REM scene3/figure 9.2 (variety of views)
6010 DIM X(16): DIM Y(16): DIM Z(16)
6020 DIM V(16): DIM W(16): DIM L(2,24)
6030 DIM A(4,4): DIM B(4,4): DIM R(4,4)
6040 LET cube = 6500: LET drawit = 7000
6050 LET PPD = 3*VERT: LET NOV = 0: LET NOL = 0
6059 REM place two cubes in ACTUAL position.
6060 GO SUB idR3
6070 GO SUB cube
6080 LET TX = 3: LET TY = 1.5: LET TZ = 2: GO SUB tran3: GO SUB mult3
6090 GO SUB cube
6099 REM loop through variety of views.
6100 GO SUB idR3: GO SUB look3
6110 CLS : GO SUB drawit
6120 GO TO 6100
6130 RETURN
```

Listing 11.2

```
7000 REM drawit/ perspective
7001 REM IN  : PPD,NOV,NOL,X(NOV),Y(NOV),Z(NOV),L(2,NOL),R(4,4)
7009 REM put vertices in OBSERVED position.
7010 FOR I = 1 TO NOV
7020 LET XX = X(I)*R(1,1) + Y(I)*R(1,2) + Z(I)*R(1,3) + R(1,4)
7030 LET YY = X(I)*R(2,1) + Y(I)*R(2,2) + Z(I)*R(2,3) + R(2,4)
7040 LET ZZ = X(I)*R(3,1) + Y(I)*R(3,2) + Z(I)*R(3,3) + R(3,4)
7049 REM perspective projection.
7050 LET V(I) = XX*PPD/ZZ
7060 LET W(I) = YY*PPD/ZZ
7070 NEXT I
7079 REM draw lines.
7080 FOR I = 1 TO NOL
7090 LET L1 = L(1,I): LET L2 = L(2,I)
7100 LET XPT = V(L1): LET YPT = W(L1): GO SUB moveto
7110 LET XPT = V(L2): LET YPT = W(L2): GO SUB lineto
7120 NEXT I
7130 RETURN
```

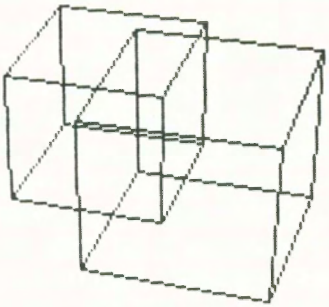


Figure 11.4

Exercise 11.2
Draw various perspective views of a wire tetrahedron and a pyramid.

The Choice of Perspective Plane

The only value required for the perspective transformation that has not yet been discussed is PPD, the distance of the perspective plane from the eye. We can see from figure 11.1 that different values of PPD produce pictures of different sizes. Which one do we choose? Is there a correct value?

If we consider the practical situation, we note that the observer is sitting in front of a television and the perspective view plane is identified with the plane of the television screen. Normally the observer is sitting at a distance that is about three times the height of the screen from the terminal. In the scale of our mapping from the real-world to the graphics area of pixels, this is a distance 3*VERT (the value we used above). If we choose PPD greater than this value it is as though we are creating a *close up*, and if PPD is less than 3*VERT we get the smaller image of a *long shot*.

Clipping

Theoretically, objects may be positioned throughout space, even behind the eye, although we consider only points with positive z-coordinates in the OBSERVED position. Even so some of these points go outside the cone of vision and become invisible. In fact, part of the cone of vision is outside the screen area (we can, after all, see the outside of the graphics area). We are left with a subset of the cone of vision, the *pyramid of vision*. Thus all points outside this pyramid (that is, those whose perspective transforms take them off the screen) must be ignored. We noted that the Spectrum displays an error message whenever we try to DRAW a line to a point off the graphics area. It is essential that we use the clipped 'lineto' routine (listing 3.4) in order to avoid any problems. In fact we further limit scenes so that all vertices in the OBSERVED position will have positive z-values; that is, all objects must lie in front of the eye (although not necessarily inside the cone of vision). This will avoid peculiar perspective projections of points that lie behind the eye appearing to be on the screen.

Exercise 11.3
Experiment with perspective views of all types of wire figures; for example, bodies of revolution, regular solids. Consider cases where an object is drawn inside the construction routine; that is, the values of V and W must now be calculated here and not in the 'drawit' routine. Change the program that drew the jet of figure 9.3 so that you get a perspective view; note that the farther the eye gets

from the plane the smaller it appears, a phenomenon that does not occur with the orthographic projection.

Exercise 11.4
Write a hidden line algorithm for a single convex body, similar to that given in chapter 10.

Note that since a convex facet is projected into a convex polygonal area in the view plane, all we need do is calculate the coordinates of the vertices of the projected facet, and hence find whether the facet is in anti-clockwise (in which case we draw it) or clockwise order (in which case it is ignored).

Exercise 11.5
Write a program that draws a perspective view of a mathematical surface, similar to that given in chapter 10. The method will be exactly equivalent to listing 10.4, with the exception that you must work with the V/W values rather than the X/Y arrays.

These hidden surface and line algorithms are perfectly adequate for specially defined single objects, but we must now consider the more general case where a number of objects are scattered about space.

Complete Programs

I. 'lib1', 'lib3' and listings 9.4 ('cube'), 11.1 ('scene3') and 11.2 ('drawit'). Data required: HORIZ, VERT, and repeated values for (EX, EY, EZ) and (DX, DY, DZ). Try 9, 6, (5, 15, 10) and (0, 0, 0); (1, 2, 20) and (0, 0, 1).

12 A General-purpose Hidden Line Algorithm

As in previous chapters, we assume that objects are set up by the 'scene3' routine, but now insist that the NOV vertices in the scene are stored in the X, Y, and Z arrays. Their perspective projections on to the view plane are stored in arrays V and W. The NOF facets are stored as a list of vertex indices (a maximum of six) in array F, and the number of edges on any facet is placed in array H.

We assume that all objects are closed. Each object need not be convex but its surface must be composed of convex facets that are stored in anti-clockwise orientation. Thus it is impossible to see the underside of any facet; that is, when projected on to the view plane we see only facets that maintain their anti-clockwise orientation. Strictly speaking, this means that we cannot draw planar objects. If these are required for a particular scene then we avoid the problem by storing each facet of a planar object twice – once clockwise and once anti-clockwise – so whatever the position of the eye, on perspective projection we see one and only one occurrence of the facet. We assume also that all lines in the scene are the edges of two contiguous facets: if a single line is required it is stored as a degenerate planar facet with two edges. These restrictions were imposed to speed up the hidden line algorithm. This is very necessary because we are now approaching the limits of the processing power of the Spectrum. Even simple pictures like the two cubes in figure 12.1 take over 5 minutes to draw, the two stars of figure 12.2 take over 30 minutes. The Spectrum was never intended to run such complex algorithms, and so it is great credit to the Sinclair design team that the Spectrum does achieve such very good results.

Nevertheless, we think it is important to study general hidden line algorithms for educational reasons. It is essential for anyone with more than a passing interest in computer graphics to understand the problems implicit in drawing views of three-dimensional objects with the hidden lines suppressed. The routine given in listing 12.1 is such a hidden line algorithm, which can be transferred to larger machines where it will run with ease. If you get the opportunity to use more powerful computers it will be very instructive to run our programs on them.

In order to produce a hidden line picture of a scene stored in the OBSERVED position, every line on the objects in the scene must be compared with every facet. Because of the above restrictions we need compare the lines only with the visible facets; that is, those that, when projected, keep their anti-clockwise orientation.

Let us assume that a typical line Γ_3 in the OBSERVED position joins the two points (x_1', y_1', z_1') and (x_2', y_2', z_2'), and so a general point on this line is

$$(1-\phi)(x_1', y_1', z_1') + \phi(x_2', y_2', z_2')$$

We suppose that these two end points are projected on to the two points (x_1, y_1) and (x_2, y_2) in the perspective view plane. Thus Γ_3 is projected on to this plane as a line Γ_2 with general point

$$(1-\mu)(x_1, y_1) + \mu(x_2, y_2)$$

Note that the point $(1-\phi)(x_1', y_1', z_1') + \phi(x_2', y_2', z_2')$ does not necessarily transform into the point $(1-\phi)(x_1, y_1) + \phi(x_2, y_2)$: that is, ϕ need not equal μ.

We let a typical facet Ω_3 be projected on to an area Ω_2 in the view plane and assume that the H vertices on this projected facet are $(\overline{x}_i, \overline{y}_i)$, where $1 \leqslant i \leqslant H$. Let the i^{th} edge of Ω_2 intersect Γ_2 at $(1-\lambda_i)(\overline{x}_i, \overline{y}_i) + \lambda_i(\overline{x}_{i+1}, \overline{y}_{i+1})$. If $\lambda_i < 0$ or $\lambda_i > 1$ then Γ_2 intersects the extended i^{th} edge at a point outside the area Ω_2: if $0 \leqslant \lambda_i \leqslant 1$ then Γ_2 crosses the area Ω_2 at a point on the i^{th} edge. Since the projected projection of a convex facet is convex, then the number of crossing points is either zero (and there is no point of intersection) or two (perhaps coincident). We need consider only the case of two non-coincident points: suppose their μ values on Γ_2 are μ_{min} and μ_{max} where $\mu_{\text{min}} < \mu_{\text{max}}$. So the points of intersection on Γ_2 are $(1-\mu_{\text{min}})(x_1, y_1) + \mu_{\text{min}}(x_2, y_2)$ and $(1-\mu_{\text{max}})(x_1, y_1) + \mu_{\text{max}}(x_2, y_2)$.

It is now necessary to discover whether the subsegment of Γ_2 between these two points is visible or not. This is checked by finding the mid-point of the segment $(x_{\text{mid}}, y_{\text{mid}}) = (1-\mu_{\text{mid}})(x_1, y_1) + \mu_{\text{mid}}(x_2, y_2)$, where $\mu_{\text{mid}} = (\mu_{\text{mid}} + \mu_{\text{max}})/2$. We then find the point $(\hat{x}, \hat{y}, \hat{z})$ on Γ_3 that has $(x_{\text{mid}}, y_{\text{mid}})$ as its perspective projection. The segment of line between the points with μ values μ_{min} and μ_{max} is hidden if and only if $(\hat{x}, \hat{y}, \hat{z})$ and the eye are on opposite sides of the infinite plane containing Ω_3. The equation of the plane is found using the methods described in chapter 7, and the functional representation of the plane is used to check the above requirement.

Note that $\hat{x}*\text{PPD}/\hat{z} = x_{\text{mid}}$, $\hat{y}*\text{PPD}/\hat{z} = y_{\text{mid}}$, and $(\hat{x}, \hat{y}, \hat{z})$ lies on Γ_3. So for some value of ϕ

$$\hat{x} = (1-\phi)x_1' + \phi x_2', \quad \hat{y} = (1-\phi)y_1' + \phi y_2', \quad \text{and} \quad \hat{z} = (1-\phi)z_1' + \phi z_2'$$

Hence

$$x_{\text{mid}} = \frac{(x_1' + \phi(x_2' - x_1'))*\text{PPD}}{z_1' + \phi(z_2' - z_1')}$$

and

$$y_{\text{mid}} = \frac{(y_1' + \phi(y_2' - y_1'))*\text{PPD}}{z_1' + \phi(z_2' - z_1')}$$

that is

$$\phi = \frac{x_{\text{mid}}*z_1' - x_1'*\text{PPD}}{(x_2' - x_1')*\text{PPD} - x_{\text{mid}}*(z_2' - z_1')}$$

$$= \frac{y_{\text{mid}}*z_1' - y_1'*\text{PPD}}{(y_2' - y_1')*\text{PPD} - y_{\text{mid}}*(z_2' - z_1')}$$

This enables us to calculate ϕ, and hence $(\hat{x}, \hat{y}, \hat{z})$, which in turn is used to find whether the subsegment Γ_2 is visible or not.

The algorithm given in routine 'hidden', listing 12.1, compares each line on the objects with all the visible (anti-clockwise) facets. Note that all objects are considered solid; that is, no individual planar facet occurs in such a way that it is possible to see its underside (clockwise orientation). The lines are implicitly stored in the facet data, and each line occurs twice, once from vertex IV1 to vertex IV2 (say) and once from vertex IV2 to IV1. Rather than duplicate effort we consider only the case when IV1 < IV2. Furthermore we compare the lines with visible facets only, for if a line is partially obscured by a hidden facet (one with clockwise orientation) then the invisible part of that line must also be obscured by anti-clockwise facets.

Let us assume that at the time of comparing the line Γ_2, which joins vertices IV1 to IV2 with the K$^{\text{th}}$ facet, we have calculated NRL visible subsegments of the line: NRL is assumed to be less than 50. The μ values for the end points of the M$^{\text{th}}$ visible segment are stored in array L at L(1,M) and L(2,M). Initially NRL = 1, L(1, 1) = 0 and L(2, 1) = 1; that is, the complete line is assumed to be visible. If at this stage a new hidden segment is discovered, specified by the values $\mu_{\min}$ and $\mu_{\max}$ (the variables MIN and MAX), then the values in the L array and NRL have to be adjusted accordingly.

When the line has been compared with all visible facets we are left with the NRL visible segments that can then be drawn on the screen. If at any time NRL becomes zero then the line is totally obscured and there is no need to continue with comparisons with other facets.

Example 12.1

We can now draw a hidden line, perspective view of the scene we first saw in figure 9.2: one of the two cubes shown in figure 12.1. The scene has HORIZ = 9, VERT = 6 and is viewed from (15, 10, 5) to (0, 0, 0).

Listing 12.1

```
7000 REM hidden/general hidden line algorithm
7001 REM IN  : NOV,NOL,X(NOV),Y(NOV),Z(NOV),V(NOV),W(NOV),
        F(6,NOF),H(NOF),PPD,R(4,4)
7010 DIM L(2,50): DIM G(NOF): LET EPS = 0.000001
7018 REM check on I'th projected facet : anticlockwise set G(I)=1
7019 REM clockwise or degenerate set G(I)=0.
7020 FOR I = 1 TO NOF
7030 LET I1 = F(1,I): LET X1 = V(I1): LET Y1 = W(I1)
7040 LET I2 = F(2,I): LET X2 = V(I2): LET Y2 = W(I2)
7050 LET I3 = F(3,I): LET X3 = V(I3): LET Y3 = W(I3)
7060 LET DX1 = X2 - X1: LET DY1 = Y2 - Y1
7070 LET DX2 = X3 - X2: LET DY2 = Y3 - Y2
7080 LET G(I) = 0
7090 IF DX1*DY2 - DX2*DY1 > 0 AND H(I) > 2 THEN LET G(I) = 1
7100 NEXT I
7109 REM find J'th line on the edge of I'th facet.
7110 FOR I = 1 TO NOF
7120 LET HH = H(I): LET IV1 = F(HH,I)
7130 LET X1 = V(IV1): LET Y1 = W(IV1)
7140 FOR J = 1 TO HH
7150 LET IV2 = F(J,I)
7160 LET X2 = V(IV2): LET Y2 = W(IV2)
7170 IF IV1 > IV2 THEN GO TO 8160
7179 REM initialise variables.
7180 LET NRL = 1: LET L(1,1) = 0: LET L(2,1) = 1
7190 LET CA = X2 - X1: LET CB = Y1 - Y2
7200 LET CC = -X1*CB - Y1*CA
7209 REM compare this line with the K'th facet.
7210 FOR K = 1 TO NOF
7220 IF G(K) = 0 THEN GO TO 8040
7230 IF K = I THEN GO TO 8040
7240 LET IN = H(K)
7249 REM loop to find two points of intersection of projected line with
        projected facet.  These points specified by MU values MIN and MAX.
7250 LET MAX = -1: LET MIN = 2
7260 LET JV1 = F(IN,K)
7270 LET VX1 = V(JV1): LET WY1 = W(JV1)
7280 LET S1 = SGN (CA*WY1 + CB*VX1 + CC)
7290 FOR M = 1 TO IN
7300 LET JV2 = F(M,K)
7310 LET VX2 = V(JV2): LET WY2 = W(JV2)
7320 LET S2 =SGN (CA*WY2 + CB*VX2 + CC)
7330 IF S1 = S2 THEN GO TO 7500
7340 LET XE = VX1 - VX2: LET YE = WY1 - WY2
7350 LET XF = VX1 - X1: LET YF = WY1 - Y1
7360 LET DISC = CA*YE + CB*XE
7369 REM if line is parallel to a line on the facet then exit facet loop.
7370 IF ABS DISC > EPS THEN GO TO 7440
7380 IF ABS CA > EPS THEN GO TO 7410
7390 IF ABS XF < EPS THEN GO TO 8040
7400 GO TO 7500
7410 LET LAMBDA = XF/CA
7420 IF ABS (XF + LAMBDA*CB) < EPS THEN GO TO 8040
7430 GO TO 7500
7440 LET LAMBDA = (CA*YF + CB*XF)/DISC
7449 REM if line misses K'th facet then go to next facet.
7450 IF LAMBDA < -EPS THEN GO TO 7500
7460 IF LAMBDA > 1 + EPS THEN GO TO 7500
7470 LET MU = (YE*XF - XE*YF)/DISC
7479 REM a true intersection so update MAX and MIN.
```

```
7480 IF MAX < MU THEN LET MAX = MU
7490 IF MIN > MU THEN LET MIN = MU
7500 LET S1 = S2
7510 LET VX1 = VX2: LET WY1 = WY2
7520 NEXT M
7529 REM check if intersections lie between specified endpoints of line.
7530 IF MIN > 1 THEN GO TO 8040
7540 IF MAX < 0 THEN GO TO 8040
7550 IF MAX > 1 THEN LET MAX = 1
7560 IF MIN < 0 THEN LET MIN = 0
7570 IF MAX - MIN < EPS THEN GO TO 8040
7579 REM calculate XMID and YMID.
7580 LET MID = (MAX + MIN)*0.5: LET MUD = 1 - MID
7590 LET XMID = MUD*X1 + MID*X2
7600 LET YMID = MUD*Y1 + MID*Y2
7610 LET DENOM = PPD*(X(IV2) - X(IV1)) - XMID*(Z(IV2) - Z(IV1))
7620 IF ABS DENOM < EPS THEN GO TO 7650
7629 REM calculate PHI and hence XHAT, YHAT and ZHAT.
7630 LET PHI = (XMID*Z(IV1) - PPD*X(IV1))/DENOM
7640 GO TO 7670
7650 LET DENOM = PPD*(Y(IV2) - Y(IV1)) - YMID*(Z(IV2) - Z(IV1))
7660 LET PHI = (YMID*Z(IV1) - PPD*Y(IV1))/DENOM
7670 LET ZHAT = (1 - PHI)*Z(IV1) + PHI*Z(IV2)
7680 LET FACT = ZHAT/PPD
7690 LET XHAT = XMID*FACT: LET YHAT = YMID*FACT
7699 REM calculate coefficients of plane containing facet: A,B,C and D.
7700 LET JV1 = F(1,K): LET JV2 =F(2,K): LET JV3 = F(3,K)
7710 LET DX1 = X(JV1) - X(JV2)
7720 LET DX3 = X(JV3) - X(JV2)
7730 LET DY1 = Y(JV1) - Y(JV2)
7740 LET DY3 = Y(JV3) - Y(JV2)
7750 LET DZ1 = Z(JV1) - Z(JV2)
7760 LET DZ3 = Z(JV3) - Z(JV2)
7770 LET A = DY1*DZ3 - DY3*DZ1
7780 LET B = DZ1*DX3 - DZ3*DX1
7790 LET C = DX1*DY3 - DX3*DY1
7800 LET D = A*X(JV1) + B*Y(JV1) + C*Z(JV1)
7810 LET S1 = A*XHAT + B*YHAT + C*ZHAT - D
7819 REM if facet hides part of line then change the L array.
7820 IF ABS S1 < EPS THEN GO TO 8040
7830 IF ABS (SGN S1 + SGN D) < 2 THEN GO TO 8040
7840 LET MORE = NRL
7850 FOR M = 1 TO NRL
7860 LET R1 = L(1,M): LET R2 = L(2,M)
7870 IF (R1 > MAX) OR (R2 < MIN) THEN GO TO 7960
7880 IF (R1 >= MIN) AND (R2 <= MAX) THEN GO TO 7950
7890 IF (R1 < MIN) AND (R2 > MAX) THEN GO TO 7920
7900 IF (R1 < MIN) THEN GO TO 7940
7910 LET L(1,M) = MAX: GO TO 7960
7920 LET MORE = MORE + 1
7930 LET L(1,MORE) = MAX: LET L(2,MORE) = R2
7940 LET L(2,M) = MIN: GO TO 7960
7950 LET L(1,M) = -1
7960 NEXT M
7969 REM tidy up the L array.
7970 LET NRL = 0
7980 FOR M = 1 TO MORE
7990 IF L(1,M) < - EPS THEN GO TO 8020
8000 LET NRL = NRL + 1
8010 LET L(1,NRL) = L(1,M): LET L(2,NRL) = L(2,M)
8020 NEXT M
8030 IF NRL = 0 THEN GO TO 8160
```

```
8040 NEXT K
8049 REM draw visible parts of line ( if any ).
8050 FOR K = 1 TO NRL
8060 LET R1 = L(1,K): LET R2 = 1 - R1
8070 LET XP1 = X1*R2 + X2*R1
8080 LET YP1 = Y1*R2 + Y2*R1
8090 LET R1 = L(2,K): LET R2 = 1 - R1
8100 LET XP2 = X1*R2 + X2*R1
8110 LET YP2 = Y1*R2 + Y2*R1
8120 IF (ABS (XP1-XP2) < EPS) AND (ABS (YP1-YP2) < EPS) THEN GO TO 8150
8130 LET XPT = XP1: LET YPT = YP1: GO SUB moveto
8140 LET XPT = XP2: LET YPT = YP2: GO SUB lineto
8150 NEXT K
8160 LET IV1 = IV2: LET X1 = X2: LET Y1 = Y2
8170 NEXT J
8180 NEXT I
8190 RETURN
```

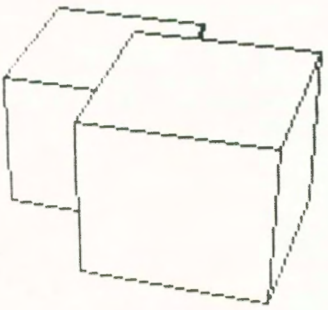


Figure 12.1

We use 'lib1', 'lib3' and 'hidden' (listing 12.1) together with the 'scene3' and 'cube' routines given in listing 12.2. This last version of 'cube' means that we have considered all the array methods of constructing an object; that is, stored/not stored, lines/facets. We deliberately used the cube over and over again in our diagrams because it is such a simple object and it is easy to understand its various constructions, and therefore it does not complicate our discussion of the general principles of three-dimensional graphics. Now is the time to introduce complexity into our objects: provided that you understand the limitations of the algorithms then the ideas we have discussed will be equally valid. Users with the 16K Spectrum will find that programs of this complexity will not fit into their machines. Such programs must be broken into independent sections and object data and arrays must be stored temporarily on tape.

Listing 12.2

```
6000 REM scene3/figure 12.1
6010 DIM X(16): DIM Y(16): DIM Z(16)
6020 DIM V(16): DIM W(16): DIM F(4,12): DIM H(12)
6030 DIM A(4,4): DIM B(4,4): DIM R(4,4): DIM Q(4,4)
6040 LET cube = 6500: LET hidden = 7000
6050 LET PPD = 3*VERT: LET NOV = 0: LET NOF = 0
6059 REM put first cube in OBSERVED position : ( ACTUAL=SETUP).
6060 GO SUB idR3: GO SUB look3
6070 GO SUB cube
6079 REM copy ACTUAL to OBSERVED matrix into Q.
6080 FOR I = 1 TO 4: FOR J = 1 TO 4
6090 LET Q(I,J) = R(I,J)
6100 NEXT J: NEXT I: GO SUB idR3
6109 REM calculate SETUP to ACTUAL matrix.
6110 LET TX = 3: LET TY = 1.5: LET TZ = 2: GO SUB tran3: GO SUB mult3
6119 REM recover ACTUAL to OBSERVED matrix.
6120 FOR I = 1 TO 4: FOR J = 1 TO 4
6130 LET A(I,J) = Q(I,J)
6140 NEXT J: NEXT I
6149 REM calculate SETUP to OBSERVED matrix.
6150 GO SUB mult3
6159 REM place second cube and draw hidden line view of scene.
6160 GO SUB cube
6170 GO SUB hidden
6180 RETURN

6500 REM cube/ vertices and facets (stored)
6501 REM IN  : PPD,NOV,NOF,X(NOV),Y(NOV),Z(NOV),V(NOV),W(NOV)
     F(4,NOF),H(NOF),R(4,4)
6502 REM OUT : NOV,NOF,X(NOV),Y(NOV),Z(NOV),V(NOV),W(NOV)
     F(4,NOF),H(NOF)
6510 DATA 1,1,1, 1,1,-1, 1,-1,-1, 1,-1,1, -1,1,1, -1,1,-1, -1,-1,-1, -1,-1,1
6520 DATA 1,2,3,4, 5,8,7,6, 1,5,6,2, 2,6,7,3, 3,7,8,4, 4,8,5,1
6530 RESTORE cube
6539 REM extend data base of vertices in OBSERVED position.
6540 LET NV = NOV
6550 FOR I = 1 TO 8
6560 READ XX,YY,ZZ: LET NOV = NOV + 1
6570 LET X(NOV) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6580 LET Y(NOV) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6590 LET Z(NOV) = XX*R(3,1) + YY*R(3,2) + ZZ*R(3,3) + R(3,4)
6599 REM perspective transform.
6600 LET V(NOV) = PPD*X(NOV)/Z(NOV)
6610 LET W(NOV) = PPD*Y(NOV)/Z(NOV)
6620 NEXT I
6629 REM read and extend data base of facets.
6630 FOR I = 1 TO 6
6640 READ F1,F2,F3,F4: LET NOF = NOF + 1
6650 LET H(NOF) = 4
6660 LET F(1,NOF) = F1 + NV: LET F(2,NOF) = F2 + NV
6670 LET F(3,NOF) = F3 + NV: LET F(4,NOF) = F4 + NV
6680 NEXT I
6690 RETURN
```

Listing 12.3

```
6500 REM icosa/hedron
6501 REM IN and OUT : same as cube above.
6510 DATA 0,1,T, T,0,1, 1,T,0, 0,-1,T, T,0,-1, -1,T,0, 0,1,-T
     -T,0,1, 1,-T,0, 0,-1,-T, -T,0,-1, -1,-T,0
6520 DATA 1,3,2, 1,2,4, 1,4,8, 1,8,6, 1,6,3, 2,3,5, 2,9,4, 4,12,8
     8,11,6, 3,6,7, 2,5,9, 4,9,12, 8,12,11, 6,11,7, 3,7,5, 5,10,9
     9,10,12, 12,10,11, 11,10,7, 7,10,5
6530 RESTORE icosa: LET T = (1 + SQR 5)/2
6540 LET NV = NOV
6550 FOR I = 1 TO 12
6560 READ XX,YY,ZZ: LET NOV = NOV + 1
6570 LET X(NOV) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6580 LET Y(NOV) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6590 LET Z(NOV) = XX*R(3,1) + YY*R(3,2) + ZZ*R(3,3) + R(3,4)
6600 LET V(NOV) = X(NOV)*PPD/Z(NOV)
6610 LET W(NOV) = Y(NOV)*PPD/Z(NOV)
6620 NEXT I
6630 FOR I = 1 TO 20
6640 READ F1,F2,F3: LET NOF = NOF + 1
6650 LET H(NOF) = 3
6660 LET F(1,NOF) = F1 + NV: LET F(2,NOF) = F2 + NV: LET F(3,NOF) = F3 + NV
6670 NEXT I
6680 RETURN
```

Listing 12.4

```
6700 REM cuboct/ahedron
6701 REM IN and OUT : same as cube above.
6710 DATA 0,1,1, 1,0,1, 1,1,0, 0,-1,1, 1,0,-1, -1,1,0, 0,1,-1
     -1,0,1, 1,-1,0, 0,-1,-1, -1,0,-1, -1,-1,0
6720 DATA 1,2,4,8, 1,6,7,3, 2,3,5,9, 4,9,10,12, 5,7,11,10, 6,8,12,11
6730 DATA 1,3,2, 1,8,6, 2,9,4, 3,7,5, 4,12,8, 5,10,9, 6,11,7, 10,11,12
6740 RESTORE cuboct
6750 LET NV = NOV
6760 FOR I = 1 TO 12
6770 READ XX,YY,ZZ: LET NOV = NOV + 1
6780 LET X(NOV) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6790 LET Y(NOV) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6800 LET Z(NOV) = XX*R(3,1) + YY*R(3,2) + ZZ*R(3,3) + R(3,4)
6810 LET V(NOV) = X(NOV)*PPD/Z(NOV)
6820 LET W(NOV) = Y(NOV)*PPD/Z(NOV)
6830 NEXT I
6840 FOR I = 1 TO 6
6850 READ F1,F2,F3,F4: LET NOF = NOF + 1
6860 LET H(NOF) = 4
6870 LET F(1,NOF) = F1 + NV: LET F(2,NOF) = F2 + NV: LET F(3,NOF) = F3 + NV
     : LET F(4,NOF) = F4 + NV
6880 NEXT I
6890 FOR I = 1 TO 8
6900 READ F1,F2,F3: LET NOF = NOF + 1
6910 LET H(NOF) = 3
6920 LET F(1,NOF) = F1 + NV: LET F(2,NOF) = F2 + NV: LET F(3,NOF) = F3 + NV
6930 NEXT I
6940 RETURN
```

Exercise 12.1

Construct hidden line scenes composed of cubes, tetrahedra, pyramids, octahedra, cuboctahedra, icosahedra. To help you, listings 12.3 and 12.4 give construction routines for a cuboctahedron and icosahedron. Write your own routines for an octahedron, and perhaps even more complicated objects like the rhombic dodecahedron (see Coxeter, 1974).

Listing 12.5

```
6000 REM scene3/two stars hidden lines removed
6010 DIM X(22): DIM Y(22): DIM Z(22)
6020 DIM V(22): DIM W(22): DIM F(3,36): DIM H(36)
6030 DIM A(4,4): DIM B(4,4): DIM R(4,4): DIM Q(4,4)
6040 LET star1 = 6500: LET star2 = 6700: LET hidden = 7000
6050 LET PPD = 3*VERT: LET NOV = 0: LET NOF = 0
6059 REM place first star.
6060 GO SUB idR3: GO SUB look3
6070 LET A = 6: GO SUB star1
6080 FOR I = 1 TO 4: FOR J = 1 TO 4
6090 LET Q(I,J) = R(I,J)
6100 NEXT J: NEXT I: GO SUB idR3
6109 REM place second star.
6110 LET TX = 5: LET TY = 5: LET TZ = 5: GO SUB tran3: GO SUB mult3
6120 FOR I = 1 TO 4: FOR J = 1 TO 4
6130 LET A(I,J) = Q(I,J)
6140 NEXT J: NEXT I
6150 GO SUB mult3
6160 LET A = 4: GO SUB star2
6170 GO SUB hidden
6180 RETURN
```

Listing 12.6

```
6500 REM star1
6501 REM IN and OUT : same as cube above.
6509 REM star based on a cube.
6510 DATA 1,1,1, 1,1,-1, 1,-1,-1, 1,-1,1, -1,1,1, -1,1,-1, -1,-1,-1, -1,-1,1,
     A,0,0, -A,0,0, 0,A,0, 0,-A,0, 0,0,A, 0,0,-A
6520 DATA 1,2,9,  2,3,9,  3,4,9,  4,1,9,  6,5,10, 5,8,10, 8,7,10, 7,6,10,
          2,1,11, 1,5,11, 5,6,11, 6,2,11, 4,3,12, 3,7,12, 7,8,12, 8,4,12,
          1,4,13, 4,8,13, 8,5,13, 5,1,13, 3,2,14, 2,6,14, 6,7,14, 7,3,14
6530 RESTORE star1
6540 LET NV = NOV
6550 FOR I = 1 TO 14
6560 READ XX,YY,ZZ: LET NOV = NOV + 1
6570 LET X(NOV) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6580 LET Y(NOV) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6590 LET Z(NOV) = XX*R(3,1) + YY*R(3,2) + ZZ*R(3,3) + R(3,4)
6600 LET V(NOV) = PPD*X(NOV)/Z(NOV)
6610 LET W(NOV) = PPD*Y(NOV)/Z(NOV)
6620 NEXT I
6630 FOR I = 1 TO 24
6640 READ F1,F2,F3: LET NOF = NOF + 1
6650 LET H(NOF) = 3
6660 LET F(1,NOF) = F1 + NV: LET F(2,NOF) = F2 + NV: LET F(3,NOF) = F3 + NV
6670 NEXT I
6680 RETURN
```

Listing 12.7

```
6700 REM star2
6701 REM IN and OUT : same as cube above.
6709 REM star based on a tetrahedron.
6710 DATA 1,1,1,  1,-1,-1,  -1,1,-1, -1,-1,1, -A,-A,-A, -A,A,A, A,-A,A, A,A,-A
6720 DATA 2,1,8,  3,2,8,  1,3,8,  1,2,7,  4,1,7, 2,4,7,
          2,3,5,  4,2,5,  3,4,5,  3,1,6,  4,3,6, 1,4,6
6730 RESTORE star2
6740 LET NV = NOV
6750 FOR I = 1 TO 8
6760 READ XX,YY,ZZ: LET NOV = NOV + 1
6770 LET X(NOV) = XX*R(1,1) + YY*R(1,2) + ZZ*R(1,3) + R(1,4)
6780 LET Y(NOV) = XX*R(2,1) + YY*R(2,2) + ZZ*R(2,3) + R(2,4)
6790 LET Z(NOV) = XX*R(3,1) + YY*R(3,2) + ZZ*R(3,3) + R(3,4)
6800 LET V(NOV) = PPD*X(NOV)/Z(NOV)
6810 LET W(NOV) = PPD*Y(NOV)/Z(NOV)
6820 NEXT I
6830 FOR I = 1 TO 12
6840 READ F1,F2,F3: LET NOF = NOF + 1
6850 LET H(NOF) = 3
6860 LET F(1,NOF) = F1 + NV: LET F(2,NOF) = F2 + NV: LET F(3,NOF) = F3 + NV
6870 NEXT I
6880 RETURN
```

Example 12.2

By now you will have realised that hidden line algorithms are very slow programs – we have to make a large number of comparisons. It takes at least 5 minutes to draw the two cubes of figure 12.1. This means that we are rather limited in the scope of objects we can draw. Nevertheless it is very good practice, and if you have the opportunity to use larger machines you will see that the above algorithm will work on these also, but much faster. We give a 'scene3' routine in listing 12.5 and examples of two three-dimensional star-shaped objects in listings 12.6 and 12.7 (both require a parameter A that changes the elongation of the spikes). These two 'star' routines are based on the tetrahedron and cube. Figure 12.2 was drawn with HORIZ = 48, VERT = 32, viewed from (35, 20, 25) towards (0, 0, 0).

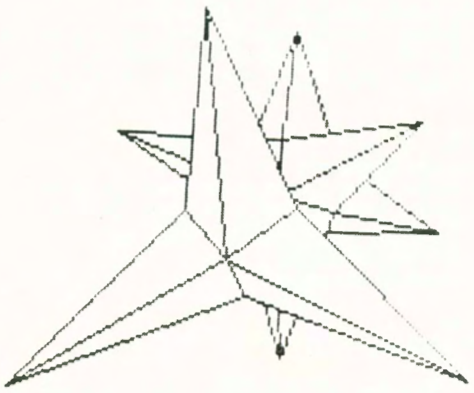


Figure 12.2

Exercise 12.2
The program in listing 10.1 checks that the order of the vertices of a triangular facet are anti-clockwise. The program was devised for use with convex bodies containing the origin. Extend it so that it can cope with the most general case; that is, specify the position of the observer and the coordinates of a point inside the object (not necessarily the origin) so that this point and the observer lie on opposite sides of the infinite plane containing the facet. Use this program to check the above star-shaped objects (in fact for these figures the origin could act as the inside point).

Then produce your own star-shaped objects based on an octahedron, cuboctahedron, icosahedron or dodecahedron. Always check the order of the vertices in your facets. You can produce stars based on very simple bodies of revolution, and we need not limit ourselves to symmetrical objects! For non-symmetrical shapes you really need the extended version of program 10.1. Provided that you stay within the restrictions mentioned, then listing 12.1 will draw any shape.

Exercise 12.3
Add extra information about the objects you are setting up. As well as the vertex and facet information, introduce another array L such that L(1,I) and L(2,I) hold the indices of the two facets that have the I^{th} line as their intersection, $1 \leqslant I \leqslant NOL$. Then change the hidden line algorithm so that it ignores any line that borders two invisible (clockwise) facets, and does not compare a facet with lines that lie in that facet.

Now you have read (and understood) chapters 7 to 12 you will have found that we have reached the limits of three-dimensional graphics on the Spectrum. You must have access to larger computers if you wish to go further in your study of this type of computer graphics. Moreover, you must study the techniques of using data structures, as opposed to arrays, for setting up scenes. For example, a complete scene can be regarded as a linked list of pointers, each of which refers to a linked list of information about the facets on a particular type of object. The facets themselves can be stored as lists of vertices! A seemingly complex idea, but one that makes the context checking of such programs as hidden line algorithms very much simpler. The relationships between objects and facets are implicitly stored in the lists. When you have grasped these ideas you can go on to the complicated graphics algorithms including methods for animating, colouring and shading. We recommend books by Horowitz and Sahni (1976), and Knuth (1972, 1973, 1981) for the study data structures, and by Newman and Sproull (1973) for the really complex graphics methods. In the next chapter we take a look at more advanced character graphics and introduce one method of producing animated three-dimensional drawings.

Complete Programs

I. 'lib1', 'lib3', listings 12.1 ('hidden') and 12.2 ('scene3' and 'cube'). Data required: HORIZ, VERT, (EX, EY, EZ), (DX, DY, DZ). Try (a) 9, 6, (15, 10, 5), (0, 0, 0); (b) 9, 6, (−10, 10, −10), (0, 1, 0).

II. 'lib1', 'lib3', listings 12.1 ('hidden') and 12.2 ('scene3'). MERGE listing 12.3 ('icosa') and 12.4 ('cuboct') and change the 'scene3' routine as follows

```
6010 DIM X(24): DIM Y(24): DIM Z(24)
6020 DIM V(24): DIM W(24): DIM F(4, 34): DIM H(34)

6040 LET cuboct=6700: LET icosa=6500: LET hidden=7000

6070 GO SUB icosa

6160 GO SUB cuboct
```

Data required: HORIZ, VERT, (EX, EY, EZ), (DX, DY, DZ). Try (a) 9, 6, (15, 10, 5), (0, 0, 0); (b) 9, 6, (−10, 10, 10), (0, 1, 0).

III. 'lib1', 'lib3', listings 12.1 ('hidden'), 12.5 ('scene3'), 12.6 ('star1') and 12.7 ('star2'). Data required: HORIZ, VERT, (EX, EY, EZ), (DX, DY, DZ). Try (a) 60, 40, (10, 20, 30), (0, 0, 0); (b) 90, 60, (−10, 20, −10), (0, 1, 0).

13 Advanced Programming Techniques

To give your programs that really professional quality it is essential to make them *user friendly*. This is one of the few pieces of advertising jargon that actually bears any relation to reality: it is essential to make programs easy to use, not just for yourself but for other people. We have all returned to programs written in a hurry three months previously, only to find that they are so badly structured/commented that we cannot understand them. It is good programming practice to comment on your programs as well as making their output self-explanatory. Ensure that the *prompts* displayed while you are actually RUNning the program are clear and concise. Another simple way of providing help is to include an introductory instruction routine as found on most video games.

In programs where a set of routines can be used in any sequence or combination, the usual method of providing for selection between options is the *menu* (see the CHARACTER GENERATOR program in chapter 5). Provided that the prompts are appropriate to the actions they initiate, this method is especially useful for people who do not understand the details of the program and are using it only as a drawing tool. Common sense plays its part in deciding what prompts should be issued. Avoid such classic misprompts as PRESS 1 FOR DUPLICATE DATA OR 2 FOR SINGLE DATA. If possible use cursor keys for movements about the screen (see option 3 of the CHARACTER GENERATOR program in chapter 5): this will seem natural to any regular user of the Spectrum. Listing 13.1 shows a program that might be used to draw polygons (as in exercise 1.3). The 'input' routine uses the 'EDIT' and cursor keys and is designed for people familiar with programming on the Spectrum.

Control Codes

The INPUT command can be used to issue a prompt immediately prior to any request for a data item. In most cases we just place the prompt, enclosed in quotes, in the exact form we wish to be displayed, before the item in the INPUT statement. On the Spectrum we can force the evaluation of any string expression, including function calls, by placing brackets around the expression. In this way it will be treated as though it were a prompt in quotes. This method is used simply in the above listing but we can produce far more complex prompts. By building

Listing 13.1

```
 10 DIM X(10): DIM Y(10)
 20 LET input = 200: LET list = 300
 30 GO SUB input
 40 PLOT X(10),Y(10): LET OX = X(10): LET OY = Y(10)
 50 FOR I = 1 TO 10: DRAW X(I) - OX,Y(I) - OY
    : LET OX = X(I): LET OY = Y(I): NEXT I
 60 PAUSE 250: GO TO 30

200 REM input coords
210 LET I = 1
220 GO SUB list
230 LET I$ = INKEY$: IF I$ = "" THEN GO TO 230
240 IF I$ = CHR$ 10 AND I < 11 THEN LET I = I + 1: GO TO 220
250 IF I$ = CHR$ 11 AND I > 1 THEN LET I = I - 1: GO TO 220
260 IF I$ <> CHR$ 7 THEN GO TO 230
270 IF I = 11 THEN INPUT "End. ? "; LINE I$: IF I$ = "y" THEN CLS : RETURN
280 IF I = 11 THEN GO TO 220
290 INPUT "X Coord. ";X(I),"Y Coord. ";Y(I): GO TO 220

300 REM list data
310 CLS : FOR J = 1 TO 10
320 PRINT AT J,1;"X Coord. ";X(J),"Y Coord. ";Y(J)
330 NEXT J: PRINT AT 11,1;"End.": PRINT AT I,0;">"
340 RETURN
```

up a string containing control codes we can display complicated coloured graphics anywhere on the screen. A named variable string can even be built up in several stages and the variable name placed in the INPUT command inside brackets. Control CODEs may be included in the strings using the CHR$ function, and numeric variables can be included by using STR$. This technique is shown to good effect in the MASTER MIND program (listing 5.6) and also in the WORM GAME (listing 1.16). Strings with built-in colour CODEs or even PRINT AT CODEs can be used in games or other programs to provide rapidly changing displays on all parts of the screen. For example, a sample menu of a mythical file-handling system is given in listing 13.2. Note the obvious use of colours to warn of potentially dangerous options.

Control codes may be entered from the keyboard (see page 115 of the Spectrum BASIC Handbook (Vickers, 1982)) for direct inclusion in strings. These direct CODEs, which are ignored in execution, can be included also in program statements to emphasise or hide parts of listings. On our accompanying tape we give program listings in which the REM statements at the start of each routine begin with the CODEs for BRIGHT 1 (extended “9”) and end with BRIGHT 0 (extended “8”). To pad out REM statements so that they end exactly at the edge of the screen we need to insert a CODE 6 in the line, which is equivalent to a comma in PRINT statements. Although not obviously available from the keyboard, we can achieve the desired effect by inserting the CODEs for PAPER 6 (extended “6”) and then immediately press DELETE. This removes the CODE for PAPER but leaves the CODE 6, which has the effect of making the cursor leap ahead to the next half or full line position.

Listing 13.2

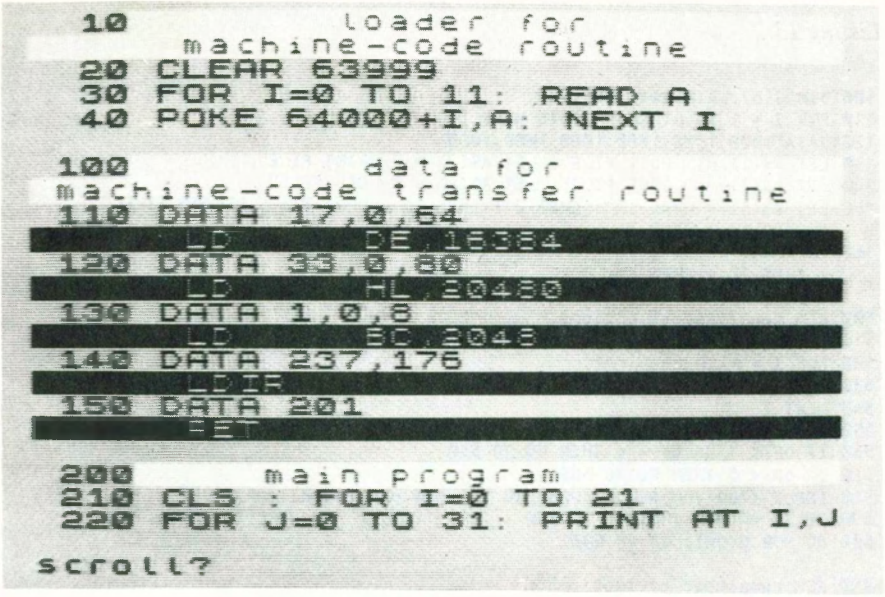
```
100 DIM G(6): DIM A$(6,24)
110 FOR I = 1 TO 6: READ G(I): NEXT I
120 DATA 1000,1000,1000,1000,1000,1000
130 LET A$(1) = " EDIT FILE": LET A$(2) = " PRINT FILE"
140 LET A$(3) = " SAVE FILE": LET A$(4) = " LOAD FILE"
150 LET A$(5) = CHR$ 17 + CHR$ 2 + CHR$ 16 + CHR$ 7 + " DELETE FILE "
    + CHR$ 17 + CHR$ 7
160 LET A$(6) = CHR$ 17 + CHR$ 2 + CHR$ 16 + CHR$ 7 + " ERASE ALL FILES "
    + CHR$ 17 + CHR$ 7

500 REM menu/ select options
510 CLS : PRINT AT 2,8;"N.O.N. FILE HANDLER"
520 FOR I = 1 TO 6
530 PRINT AT I*2 + 4,8;I;"   ";A$(I)
540 NEXT I
550 INPUT "SELECT OPTION ?";OP
560 IF OP < 1 OR OP > 6 THEN GO TO 550
570 IF OP < 5 THEN GO TO 600
580 INPUT ("DO YOU REALLY WANT TO ";+CHR$ 6 + A$(OP) + " ");Y$
590 IF Y$ <> "y" THEN GO TO 550
600 GO SUB G(OP): GO TO 500

1000 REM remainder of routines
1010 RETURN
```

Here CODE 6 is interpreted in its own right and not as a parameter following the CODE for PAPER. A CODE 6 at the end of each line preceding a new section of program can be used to force the display of a blank line in the listing. The word REM can be hidden by placing CODEs for INK 0 (extended shift "0") after REM and INK 7 (extended shift "7") before it.

As a rule it is best to write programs in modules. Highlighting the names of the modules allows us to read, correct and adapt the programs with ease. To demonstrate the improved legibility provided by these highlighting techniques see figure 13.1, which shows part of listing 13.4 as it appears on the screen.

Structure of the Display File

We know that the display file can be directly altered or examined by BASIC programs, but to make use of this facility we must understand how the display file is organised. Each horizontal line on the display is made up of 32 bytes, each of eight bits. These 32 bytes are stored in consecutive locations in the memory. However the bytes for the next line down are held in the equivalent position on the next page of memory. For an eight-bit processor, like the Z80 contained in the ZX Spectrum, one *page* of memory is 2^8 or 256 bytes: we shall use the identifier PAGE to refer to this quantity. This arrangement is designed to help the video circuitry cope efficiently with the display. Thus each page has room for eight lines of 32 bytes. The first page of the display file contains the top line for each of the first eight rows of character blocks. To complete the



```
  10           loader for
     machine-code routine
  20 CLEAR 63999
  30 FOR I=0 TO 11: READ A
  40 POKE 64000+I,A: NEXT I

 100             data for
machine-code transfer routine
 110 DATA 17,0,64
        LD        DE,16384
 120 DATA 33,0,80
        LD        HL,20480
 130 DATA 1,0,8
        LD        BC,2048
 140 DATA 237,176
        LDIR
 150 DATA 201
        RET

 200        main program
 210 CLS  : FOR I=0 TO 21
 220 FOR J=0 TO 31: PRINT AT I,J

scroll?
```

Figure 13.1

remaining seven lines for these rows there are seven more pages of memory, each page containing the data for subsequent lines. After these eight pages there are another two sets of eight pages holding the line information for the rest of the screen. From this we can calculate which bytes holds the data for the top line of a character block, and know that each of the other lines down the block is stored 256 bytes (one page) farther on. To work out the display-file position of the top line of a character block we need to know in which third of the screen (eight pages) the block lies. We need also to identify which of the 256 blocks in this third we are considering. We calculate the position of the first line of a character block in row R, column C by the following FuNction

DEF FN A(R,C) = 16384 + INT (R/8)*2048 +(R – INT(R/8)*8)*32 + C

This function is made up of four parts

16384 (to the start of the display file)
+ INT (R/8)*2048 (add relevant third of screen (0, 1, 2) multiplied by the length of one-third of the screen (2048 = 8*PAGE))
+ (R – INT(R/8)*8)*32 (plus the position of the row within that third of the screen (0-7) multiplied by 32 (columns per row))
+ C (plus the column (0-31) within that row)

The first two items determine the location of the start of the page that holds the first lines of that particular third of the screen. The last two items establish which of the 256 bytes along the page corresponds to the top line of the character block.

Using another function with a character as the input parameter, we can find the start of the data that define this character and then transfer these data to the display-file locations

DEF FN T(A$)= PEEK 23606 + PEEK 23607*256 + 8*CODE A$

The above function uses the system variable CHARS (stored in locations 23606 and 23607, see chapter 5) to find the start of the table of character set data. It then adds eight times the CODE for the required character and thus finds the address of the first piece of data. Note that these two functions can be used to write on the bottom two lines of the display, which are not normally accessible, as is demonstrated by the program in listing 13.3.

Listing 13.3

```
100 REM main program
110 LET print = 500
120 LET P$ = "9 FAKE statement, 120:1": LET ROW = 23: LET COL = 0
130 GO SUB print
140 LET P$ = "print this anywhere on the screen": LET ROW = 7: LET COL = 16
150 GO SUB print
160 PAUSE 250: STOP

500 REM print routine
510 DEF FN A(R,C) = 16384 + INT (R/8)*2048 + (R - INT (R/8)*8)*32 + C
520 DEF FN D(A$) = PEEK 23606 + 256 * PEEK 23607 + 8*CODE A$
530 FOR I = 1 TO LEN P$
540 LET ADDRESS  = FN A(ROW,COL)
550 LET DATA1 = FN D(P$(I))
560 FOR J = 0 TO 7
570 POKE ADDRESS + J*256, PEEK (DATA1 + J)
580 NEXT J
590 LET COL = COL + 1: IF COL  = 32 THEN LET COL = 0
    : LET ROW  = ROW + 1 : IF ROW = 24 THEN LET ROW = 0
600 NEXT I
610 RETURN
```

Rapid Transfer of Screen Data

If we try to move the display file of the Spectrum about rapidly, we soon find that BASIC is not fast enough to transfer the required quantity of data adequately. The Spectrum is controlled by a Z80 microprocessor that is ideally suited to the task of moving data about quickly. To program this directly (instead of using BASIC as an intermediate language) involves an excursion into *machine-code*. The Z80 machine-code has instructions, like the PEEK and POKE of BASIC, which can be used to examine or replace numbers in various locations.

The Z80 also has internal *registers*, in which numbers can be placed, and used in much the same way as variables in BASIC. So we can write a set of machine-code instructions (that is, a program) to load one of these registers with a number from a memory location, and then load the number from the register into another memory location. This may not sound very interesting until you realise that the Z80 chip can carry out this program about a million times in one second! Unfortunately machine-code appears rather incomprehensible to a beginner, looking like a stream of apparently unrelated numbers. It is more convenient to use *assembly language*, which contains short words, *mnemonic codes*, that are a great help in understanding the function of each machine-code instruction. The Spectrum requires machine-code instructions to be stored in DATA statements and then POKEd into the memory. It is helpful to include the assembly language instructions as a REMark in the DATA line. *Assemblers* are programs that translate assembly language into machine-code, but if an assembler is unavailable we can use the table given in appendix A, page 183, of the Spectrum BASIC Handbook (Vickers, 1982). One of the most powerful instructions available on the Z80 chip is 237, 176 or, more comprehensibly, **LDIR** in assembly language. This stands for **L**oa**D** and **I**ncrement **R**epeated, and is an instruction used to transfer blocks of data from one place to another in the store. Before issuing this instruction, however, we must load some of the registers with various information. We must load

DE with the address of the destination for the data
HL with the current address of the data
BC with the number of bytes to be transferred

For clarity we look at figure 13.2, an example machine-code routine, with the equivalent Assembly language, and also a BASIC program, which performs exactly the same function. Remember the Assembly language is just a way of making machine-code more understandable. The complete program, listing 13.4, POKEs the machine-code into the memory and then calls the routine. This example copies the bottom third of the display file into the top third of the screen.

```
                                 300 REM BASIC data transfer
17,0,64    LD     DE,16384       310 LET DE = 16384
33,0,80    LD     HL,20480       320 LET HL = 20480
1,0,8      LD     BC,2048        330 LET BC = 2048
237,176    LDIR                  340 LET A = PEEK HL: POKE DE,A
                                 350 LET DE = DE + 1: LET HL = HL + 1
                                 360 LET BC = BC – 1
                                 370 IF BC <> 0 THEN GO TO 340
201        RET                   380 RETURN
```

Figure 13.2

Having POKEd the machine-code instructions into the memory we run the routine by using the USR function with the start address of our routine. USR with a numerical address simply calls the subroutine, in machine-code, starting at that address. This call is executed in a BASIC program by a command like LET A = USR 32000 (32000 is the address holding 17, the first byte of code).

Listing 13.4

```
 10 REM Loader for machine-code routine
 19 REM for 16K machines use clear 31999.
 20 CLEAR 63999
 30 FOR I = 0 TO 11: READ A
 39 REM for 16K use POKE 32000.
 40 POKE 64000 + I,A: NEXT I

100 REM data for machine-code transfer routine
110 DATA 17,0,64                        : REM  LD     DE,16384
120 DATA 33,0,80                        : REM  LD     HL,20480
130 DATA 1,0,8                          : REM  LD     BC,2048
140 DATA 237,176                        : REM  LDIR
150 DATA 201                            : REM  RET

200 REM main program
210 CLS : FOR I = 0 TO 21
220 FOR J = 0 TO 31: PRINT AT I,J;CHR$ (I + 64): NEXT J
230 NEXT I
240 PRINT AT 19,1; INVERSE 1;"PRESS ANY KEY TO START ROUTINE"
250 IF INKEY$ = "" THEN GO TO 250
259 REM for 16K change to USR 32000.
260 LET A = USR 64000: STOP
```

Exercise 13.1
Type in the BASIC routine from figure 13.2 and time how long it takes to run. Compare this with the time taken for the machine-code routine in listing 13.4. You will find the machine-code is thousands of times faster.

Animation (48K machines only)

We use a simple routine to transfer all the data necessary to display a picture (both display file and attribute file) from somewhere else in the memory to the screen memory locations. This provides a quick method of changing between pictures. There is enough memory on the 48K Spectrum to hold five alternative pictures or *frames*. We can use the program from listing 13.5 to construct five machine-code routines at 30100, 30200, 30300, 30400 and 30500 that will transfer these pictures to the screen. Suppose five diagrams have been SAVEd on tape. We LOAD them into the alternative frames in memory, from where they can be copied on to the screen by calling the appropriate routine. This method could be used to illustrate a lecture or a sales talk with a short slide show, switching almost instantaneously between slides. Naturally after five slides have been shown then five more must be LOADed. This takes approximately 5 minutes from tape but less than 10 seconds from disk.

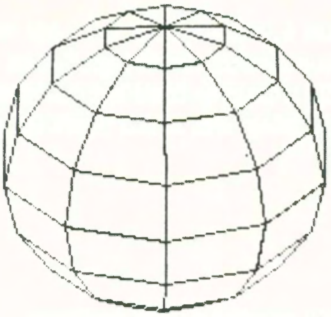
Pressing one of the keys "1" to "5" when using the routine 'slideshow' (listing 13.5) brings the required picture to the screen.

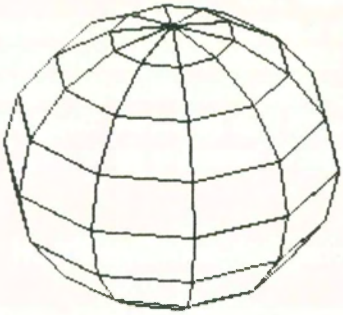
Listing 13.5

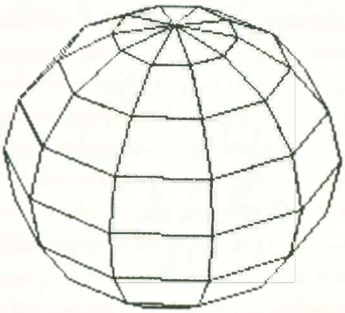
```
100 REM loader for machine-code routines
110 CLEAR 29999: DIM F(5)
120 FOR I = 1 TO 5: RESTORE
130 LET F(I) = 120 + 27*(I - 1)
140 FOR J = 0 TO 11
150 READ A: POKE 30000 + I*100 + J,A
160 NEXT J
170 NEXT I

200 REM data transfer from I'th frame to screen
210 DATA 17,0,64                        : REM  LD     DE,16384
220 DATA 33,0,F(I)                      : REM  LD     HL,FRAME(I)
230 DATA 1,0,27                         : REM  LD     BC,27*256
240 DATA 237,176                        : REM  LDIR
250 DATA 201                            : REM  RET

300 REM load frames
310 FOR J = 1 TO 10: BEEP 0.1,30: PAUSE 5: NEXT J
320 CLS : FOR I = 0 TO 4
330 LOAD "" CODE (120 + I*27)*256,27*256
340 NEXT I
350 FOR J = 1 TO 10: BEEP 0.1,30: PAUSE 5: NEXT J

400 REM slide show
410 IF INKEY$ <> "" THEN GO TO 410
420 LET A$ = INKEY$: IF A$ = "" THEN GO TO 420
430 IF A$ = CHR$ 13 THEN GO TO 310
440 IF A$ = "m" THEN GO TO 500
450 IF A$ >= "1" AND A$ <= "5" THEN LET A = USR (30000 + VAL A$*100)
460 GO TO 410

500 REM movie show
510 FOR I = 30100 TO 30500 STEP 100
520 LET A = USR I
530 IF INKEY$ = "s" THEN GO TO 400
540 NEXT I: GO TO 510
```

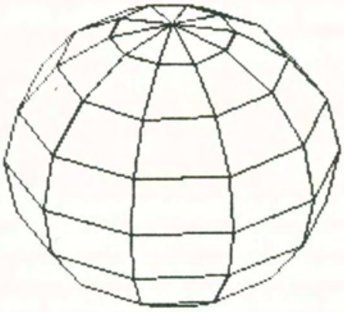
This idea can be extended to produce 'movie's. For example, consider figure 13.3, which shows five frames produced by the three-dimensional routines of chapter 11. These give views of the same spheroid are created using HORIZ = 3.2, VERT = 2.2, NUMH = 10, NUMV = 8 and PHI = 0.4 * PI * I/NUMH, where $0 \leqslant I \leqslant 4$. The 'movie' of this object, viewed from (1, 2, 3) to (0, 0, 0), can be shown apparently rotating by repeatedly bringing the five frames on to the screen in quick succession. This set of pictures is included on the tape and is shown below.

After you have finished with the 'movie' or 'slideshow' program you should type CLEAR 65367, for otherwise the next time you LOAD a program the Spectrum will reply 'Out of Memory'.


(a)


(b)


(c)


(d)

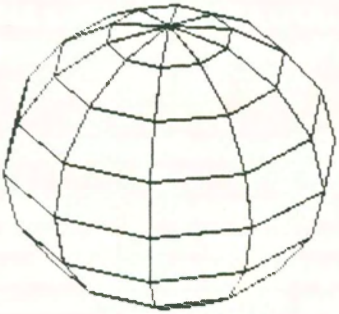

(e)

Figure 13.3

Exercise 13.2
Draw a perspective view of a wire cube in frame 1, and the same view, but with the hidden lines suppressed, in frame 2. Produce a movie consisting of these two frames only. You will see the visible lines of the cube stay fixed but the hidden lines will flash on and off. Construct a movie in which one of the 'star's from chapter 12 appears to rotate.

Scrolling (16K and 48K machines)

Using machine-code for more complicated tasks requires a great deal more thought and study. A good book on the subject and preferably one specifically written for the Spectrum should be consulted (see Zaks, 1978; Hutty, 1981; Woods, 1983). For those who wish to pursue more complex manipulations of the screen in machine-code we give one more example. (Listing 13.6 scrolls the graphics display area downwards. The fact that the display-file data are split into lines of 32 bytes causes problems. To effectively move these data around we need the start addresses of these lines. To calculate the addresses in a machine-code routine requires a lot of programming effort and slows down the execution. Instead we use a *look-up table*. For machine-code purposes an address is made up of sixteen bits that are stored in two eight-bit locations, the *lo-byte* and the *hi-byte*. Whenever we need to know the address of a line we simply look up its two halves from the tables. We use BASIC to calculate the tables and to POKE them into memory. These tables and the machine-code itself could be saved on tape and reloaded if required. Listing 13.6 gives the machine-code program, BASIC loader and table constructer. Listing 13.7 shows an equivalent BASIC routine for the program. Note that the BASIC routine assumes the existence of the same table used by the machine-code.

Exercise 13.3
Write a machine-code routine that moves the attribute file around the memory, one row at a time. Then write a BASIC program that calls your routine each time the graphics area has been scrolled down by eight lines.

BASIC Structure (Renumber and Delete)

The development of modular programs often leads to situations where routines are cramped and there is no room for extra lines or alterations. At times like these we would like to *renumber* the lines automatically or perhaps *delete* whole sections. These two utility commands are also useful when MERG(E)ing programs in order to create new routines. To perform these tasks we must take a closer look at the way our BASIC programs are stored in the computer memory.

BASIC programs are stored line by line in memory starting at address 23755.

Listing 13.6

```
 10 REM loader for machine-code routine
 20 CLEAR 31999
 30 FOR I = 0 TO 74
 40 READ A: POKE 32000+I,A
 50 NEXT I

100 REM downwards wrap-around scroll for graphics area
110 DATA 1,175,0                      : REM LD    BC,175
120 DATA 221,33,75,125                : REM LD    IX,LOBYTE
130 DATA 221,9                        : REM ADD   IX,BC
140 DATA 221,110,0                    : REM LD    L,[IX+0]
150 DATA 221,33,251,125               : REM LD    IX,HIBYTE
160 DATA 221,9                        : REM ADD   IX,BC
170 DATA 221,102,0                    : REM LD    H,[IX+0]
180 DATA 17,171,126                   : REM LD    DE,TEMP
190 DATA 205,67,125                   : REM CALL  MOVE
200 DATA 221,33,75,125                : REM LD    IX,LOBYTE
210 DATA 221,9                        : REM ADD   IX,BC
220 DATA 221,94,0                     : REM LD    E,[IX+0]
230 DATA 221,110,-1                   : REM LD    L,[IX-1]
240 DATA 221,33,251,125               : REM LD    IX,HIBYTE
250 DATA 221,9                        : REM ADD   IX,BC
260 DATA 221,86,0                     : REM LD    D,[IX+0]
270 DATA 221,102,-1                   : REM LD    H,[IX-1]
280 DATA 205,67,125                   : REM CALL  MOVE
290 DATA 13                           : REM DEC   C
300 DATA 32,226                       : REM JR    NZ,LINE
310 DATA 17,0,64                      : REM LD    DE,16384
320 DATA 33,171,126                   : REM LD    HL,TEMP
330 DATA 205,67,125                   : REM CALL  MOVE
340 DATA 201                          : REM RET
350 DATA 197                          : REM PUSH  BC
360 DATA 1,32,0                       : REM LD    BC,32
370 DATA 237,176                      : REM LDIR
380 DATA 193                          : REM POP   BC
390 DATA 201                          : REM RET

400 REM construct tables of display-file line addresses
410 FOR I = 0  TO 21: FOR J = 0 TO 7
420 LET A = 16384 + INT (I/8)*2048 + (I - INT (I/8)*8)*32
430 LET H = INT (A/256): LET L = A - H*256
440 POKE 32075 + I*8 + J,L: POKE 32251 + I*8 + J,H + J
450 NEXT J: NEXT I

500 REM main program
510 LIST 380
520 IF INKEY$ <> "" THEN LET A = USR 32000
530 GO TO 520
```

The first item stored for each line is the line number (16 bits), which takes up two locations and is stored as the hi-byte followed by the lo-byte. In all other places values are stored in the standard lo-hi format. To illustrate suppose we enter the line 10 REM. If we type

PRINT PEEK 23755, PEEK 23756

Listing 13.7

```
 10 REM basic version of scroll for graphics area
 20 CLEAR 31999
 30 GO TO 400

100 REM downwards wrap-around scroll for graphics area
110 LET C = 175: LET B = 0: LET BC = C + B*256
120 LET IXLO = 75: LET IXHI = 125: LET IX = IXLO + IXHI*256
130 LET IX = IX + BC
140 LET L = PEEK (IX + 0)
150 LET IXLO = 251: LET IXHI = 125: LET IX = IXLO + IXHI*256
160 LET IX = IX + BC
170 LET H = PEEK (IX + 0): LET HL = L + H*256
180 LET E = 171: LET D = 126: LET DE = E + D*256
190 GO SUB 350
200 LET IXLO = 75: LET IXHI = 125: LET IX = IXLO + IXHI*256
210 LET IX = IX + BC
220 LET E = PEEK (IX + 0)
230 LET L = PEEK (IX - 1)
240 LET IXLO = 251: LET IXHI = 125: LET IX = IXLO + IXHI*256
250 LET IX = IX + BC
260 LET D = PEEK (IX + 0): LET DE = E + D*256
270 LET H = PEEK (IX - 1): LET HL = L + H*256
280 GO SUB 350
290 LET C = C - 1: LET BC = C + B*256
300 IF C <> 0 THEN GO TO 200
310 LET E = 0: LET D = 64: LET DE = E + 256*D
320 LET L = 171: LET H = 126: LET HL = L + H*256L
330 GO SUB 350
340 RETURN
350 LET S = BC
360 LET BC = 32
370 LET A = PEEK HL: POKE DE,A: LET DE = DE + 1: LET HL = HL + 1
    : LET BC = BC - 1: IF BC <> 0 THEN GO TO 370
380 LET BC = S
390 RETURN

400 REM construct tables of display-file line addresses
410 FOR I = 0  TO 21: FOR J = 0 TO 7
420 LET A = 16384 + INT (I/8)*2048 + (I - INT (I/8)*8)*32
430 LET H = INT (A/256): LET L = A - H*256
440 POKE 32075 + I*8 + J,L: POKE 32251 + I*8 + J,H + J
450 NEXT J: NEXT I

500 REM main program
510 LIST 380
520 IF INKEY$ <> "" THEN GO SUB 100
530 GO TO 520
```

we shall see that 0, 10 is stored in these locations. If we remove line 10 and enter another numbered line at the start of the program, we see that the representation of this new number is now stored in 23755 and 23756. The next two bytes give the length of the present line (this can be used to find the start of the next program line without traversing the whole of the present line). Now comes the actual text of the BASIC line, each character or keyword taking up one memory location. The end of the line is marked by a CODE 13 (or ENTER). After any number given in decimal notation and stored as characters there is a CODE 14 (or NUMBER: see page 183 of the Spectrum BASIC Handbook (Vickers, 1982)) followed by a binary translation of the number, taking five bytes. Integers are stored in a simple way, as explained on page 163 of the Spectrum BASIC Handbook (Vickers, 1982). Using this information about the contents of the memory we can write a program that lists programs, including itself, by examining the memory. This type of listing program (listing 13.8) can be of great assistance when formatting listings in which statements all start on a new line, or when control CODES need to be displayed.

Listing 13.8

```
100 REM self-listing program
110 LET I = 23755: CLS
120 LET LINE = 256*PEEK I + PEEK (I + 1): IF LINE > 9999.5 THEN STOP
130 PRINT " ";LINE;
140 LET I = I + 2: PRINT PAPER 5;PEEK I;",";PEEK (I + 1);" ";
150 LET LENGTH = PEEK I + 256*PEEK (I + 1)
160 LET NUM = 0: LET I = I + 1
170 FOR J = 1 TO LENGTH: LET I = I + 1
180 LET P = PEEK I: IF P < 32 AND NUM <= 0 THEN LET NUM = 1
190 IF P = 13 THEN PRINT PAPER 4;P: PRINT: GO TO 230
200 IF P = 14 THEN LET NUM = 6
210 IF NUM > 0 THEN PRINT PAPER 6;P;",";: GO TO 230
220 PRINT CHR$ P;
230 LET NUM = NUM - 1
240 NEXT J
250 LET I = I + 1: GO TO 120
```

In order to renumber lines we need to change the line numbers stored in the memory. This is true as long as the lines do not get out of numerical sequence. Unfortunately this does not take into account GO TO or GO SUB, etc., commands. We must search out all GO SUB, GO TO, RESTORE and RUN keywords in the program and, if they are followed by a simple integer, check whether the line number has been changed. If it has, then both the numeric characters of the number and the integer representation of the number must be altered. To delete a range of lines all we have to do is extend the length of the line before the unwanted range, so that it completely covers the unwanted area. The correct length of the line is re-established on editing and re-entering this line. The remainder of the program is moved down through the store to continue from where the line now ends. Listing 13.9 gives two routines that perform these tasks; the renumber routine is entered by RUN 9900 and the delete routine by RUN 9970.

Exercise 13.4
Write a routine that can search through the BASIC memory and find any specified sequence of CODEs. Use the method shown in the above delete program to set the system variable E PPC (which controls the edit cursor: see page 174 of the Spectrum BASIC Handbook (Vickers, 1982)) to the number of the line where the sequence is found.

Listing 13.9

START = PEEK 23635 + PEEK 23636*256

NO MICRODRIVES FITTED

```
9900 INPUT "Renumber lines ";LOW;" To ";UP,"Starting at ";NEW;
     " in steps of ";STEP
9901 LET X = 9900: IF LOW > X OR UP > X OR LOW > UP THEN GO TO 9900
9902 IF NEW > X OR NEW < 1 OR STEP > X OR STEP < 1 THEN GO TO 9900
9903 PRINT AT 19,0;"Renumber lines ";LOW;" to ";UP,"Starting at ";NEW;
     " in steps of ";STEP,"Please wait"
9904 LET START = 23755: LET SCREEN = 16384: LET HI = 256
9905 LET A = START: LET B = SCREEN
9906 LET H = PEEK A: LET L = PEEK (A + 1)
9907 POKE B,H: POKE B + 1,L
9908 LET A = A + 2: LET B = B + 2
9909 LET LEN = PEEK A + PEEK (A + 1)*HI: LET A = A + LEN + 2
9910 IF H*HI + L < X THEN GO TO 9906
9911 LET A = START: LET B = SCREEN: LET W = NEW
9912 LET H = PEEK A: LET L = PEEK (A + 1)
9913 LET N = HI*H + L: IF N < LOW OR N > UP THEN GO TO 9918
9914 IF W > X THEN GO TO 9955
9915 LET H = INT (W/HI): LET L = W - H*HI
9916 POKE A,H: POKE A + 1,L
9917 LET W = W + STEP
9918 LET A = A + 2
9919 LET LEN = PEEK A + PEEK (A + 1)*HI: LET A = A + LEN + 2
9920 IF N < X THEN GO TO 9912
9921 LET A = START: LET B = SCREEN
9922 LET H = PEEK A: LET L = PEEK (A + 1): LET NN = PEEK B*HI + PEEK (B + 1)
9923 LET N = H*HI + L: IF N = X THEN STOP
9924 PRINT AT 21,0;"Checking ";NN;" = new line ";N
9925 LET A = A + 2: LET W = A + 2
9926 LET LEN = PEEK A + PEEK (A + 1)*HI: LET A = A + LEN + 2
9927 LET T = PEEK W
9928 IF T = CODE (" GO TO ") OR T = CODE (" GO SUB ") OR T = CODE (" RUN ")
     OR T = CODE (" RESTORE ") THEN GO SUB 9932
9929 IF T = 14 THEN LET W = W + 5
9930 LET W = W + 1: IF W < A THEN GO TO 9927
9931 LET B = B + 2: GO TO 9922
9932 LET V = W + 1: LET A$ = ""
9933 LET S = PEEK V: IF S <> 32 AND S <> 14 AND NOT (S>=48 AND S<=57)
     THEN RETURN
9934 IF S <> 14 THEN LET V = V + 1: LET A$ = A$ + CHR$ S: GO TO 9933
9935 LET V = V + 3: LET AT = PEEK V + PEEK (V + 1)*HI
9936 IF AT < LOW OR AT > UP THEN RETURN
9937 GO SUB 9943: LET H = INT (NAT/HI): LET L = NAT - H*HI
9938 LET B$ = STR$ NAT: IF LEN A$ <> LEN B$ THEN GOSUB 9951
9939 POKE V,L: POKE V+1,H
9940 FOR I = 1 TO LEN B$
9941 POKE W + I,CODE B$(I): NEXT I
9942 RETURN
9943 LET C = START: LET D = SCREEN
9944 LET H = PEEK D: LET L = PEEK (D + 1)
9945 LET E = PEEK  C *HI + PEEK (C + 1)
9946 IF E >= X THEN LET NAT = 0: RETURN
9947 IF H*HI + L = AT THEN LET NAT = E: RETURN
9948 LET C = C + 2: LET D = D + 2
9949 LET LEN = PEEK C + PEEK (C + 1)*HI: LET C = C + LEN + 2
9950 GO TO 9944
9951 LET DIFF = LEN A$ - LEN B$
9952 IF DIFF > 0 THEN FOR I = 1 TO DIFF: LET B$ = B$ + " ": NEXT I: RETURN
9953 PRINT AT 16,0;"No room at ";CHR$ T;AT;" in line ";NN,
     "type edit and add ";-DIFF;" space(s)","to label then re-run program."
9954 LET H = INT (NN/HI): POKE 23626,H: POKE 23625,NN-H*HI
```

```
9955 LET A = START: LET B = SCREEN
9956 LET H = PEEK A: LET L = PEEK (A+1)
9957 IF H*HI + L = X THEN PRINT AT 0,0;"Renumber aborted": STOP
9958 POKE A, PEEK B: POKE A+1, PEEK (B + 1)
9959 LET A = A + 2: LET B = B + 2
9960 LET LEN = PEEK A + PEEK (A + 1)*HI: LET A = A + LEN + 2
9961 GO TO 9956
9970 INPUT "Delete lines ";LOW;" to ";UP: IF LOW < 2 OR LOW >= UP
     THEN GO TO 9970
9971 PRINT AT 20,0;"Delete lines ";LOW;" to ";UP,"Please wait"
9972 LET START = 23755: LET SCREEN = 16384: LET HI = 256
9973 LET A = START: LET B = SCREEN
9974 LET H = PEEK A: LET L = PEEK (A + 1)
9975 IF H*HI + L >= LOW THEN PRINT AT 0,0;"Please enter line ";
     LOW-1;" REM ","and re-run program": STOP
9976 LET H = PEEK A: LET L = PEEK (A + 1)
9977 IF  H*HI + L >= LOW THEN GO TO 9981
9978 LET NN = H*HI +L: LET LAST = A: LET A = A +2
9979 LET LEN = PEEK A + PEEK (A + 1)*HI: LET A = A + LEN + 2
9980 GO TO 9976
9981 LET LAST = LAST + 2
9982 LET LONG = PEEK LAST + PEEK (LAST + 1)*HI
9983 IF H*HI + L > UP THEN PRINT AT 0,0;"No lines in range": STOP
9984 LET H = PEEK A: LET L = PEEK (A + 1): IF H*HI + L = UP THEN GO TO 9990
9985 LET A = A + 2: LET LONG = LONG + 2
9986 LET LEN = PEEK A + PEEK (A + 1)*HI: LET A = A + LEN + 2
     : LET LONG = LONG + LEN + 2
9987 IF H*HI + L <> UP THEN GO TO 9984
9990 LET H = INT (LONG/HI): POKE LAST + 1,H: POKE LAST,LONG - H*HI
9995 LET H = INT (NN/HI): POKE 23626,H: POKE 23625,NN-H*HI
9999 PRINT AT 0,0;"type edit and enter": STOP
```

BASIC Structure (Efficient Programs)

When programming in BASIC we can use our knowledge of the way lines are stored to help make our programs more efficient, both in terms of space used and speed of execution. Every occurrence of an explicit number is followed by six bytes of number codes so that the conversion from a string of digits to its binary form need not be performed each time the line is entered. If instead we assign the required number to a variable, and use the variable name for each occurrence of the value, then the space required would be only the number of bytes in the variable name. To assign the value we must use three extra codes ":", "LET" and "=", as well as the variable name, and to access the value we must use the variable name. If the value is used often then this method will save considerable space. As an example consider the first two statements in the following program

```
10 PRINT AT 1,1
20 LET A = 1: PRINT AT A,A
30 PRINT AT 1,1
40 PRINT AT A,A
```

The BASIC text of line 10 requires 17 bytes of store: "PRINT", "AT", "1", 14, 0, 0, 1, 0, 0, ",", "1", 14, 0, 0, 1, 0, 0.

Line 20 requires 16 bytes of store: "LET", "A", "=", "1", 14, 0, 0, 1, 0, 0, ":", "PRINT", "AT", "A", ",", "A".

The second line above has fewer codes than the first line. However, the second line uses the variable A, which, unless it is used elsewhere, will take up six bytes of memory. If we need to repeat the statements, as shown above in lines 30 and 40, we find that line 30 is 17 bytes long but line 40 is only 5 bytes long. So if we use a numeric value more than a couple of times in a program, we can make large savings by using a variable to represent the value.

We can also save space and improve speed by packing as many statements as possible on to one line, although this reduces legibility. Each new line requires five bytes of memory: one for the CODE 13 at the end of the previous line, two for the new line number and two for the length of the line. A colon between statements needs only one byte. The space saved can be quite large. For example, consider a program with 90 statements (quite small on average): if the statements are initially all on separate lines and we rewrite with 3 statements per line, we can save 60*(5 − 1) = 240 bytes. In a large program it is quite feasible to save over 1K of memory in this way. Using fewer lines also makes it easier for the machine to obey GO TO or GO SUB commands, as it will take less time to search for the destination line. To be most efficient all SUBroutines and FuNctions should be near the start of a program and arranged so that the most frequently used come first. Unless very many calls are made to the same routine, it is usually more convenient to store the routines in a logical order.

Exercise 13.5
Rewrite and reposition the graphics routines and/or the games programs to use less space and if possible run faster.

Synchronous Display

When displaying pictures on the Spectrum we can use the PAUSE command to provide an interesting BORDER display. The television display is completely redrawn every 1/50th of a second (1/60th in the U.S.A.) and PAUSE uses multiples of this same time interval. When PAUSE 1 is used the delay is not always a complete 50th of a second. In fact the PAUSE will last only until the start of the next frame. This gives us a method of starting instructions at a fixed time relative to each refresh of the display. If we change the BORDER colour partway through the drawing of the display, then for that frame the top part only of the BORDER will be one colour and the remainder will be another colour. Obviously we could use PAUSE to wait for the start of the next frame and repeat the process. This results in the display of a steady picture provided we keep repeating the same actions. The SAVE and LOAD statements use a machine-code routine to do an equivalent process, which is how the red/cyan and yellow/blue patterns are produced on the BORDER.

Consider the following start of a large program

```
1 GO TO 10
2 PAUSE 1: BORDER 7: BORDER 2: BORDER 6: BORDER 4:
BORDER 5: BORDER 1: BORDER 3: BORDER 7: GO TO 2
```

The program terminates with a GO TO 2 command that produces a rainbow effect on the BORDER after the construction of a diagram. Note that, unless the line is placed near the top of the program, the GO TO statement will take so long to execute that we shall miss the start of the next frame.

Exercise 13.6

Write a one-line program that alternates between red and cyan BORDER colours without using PAUSE. Insert extra colons (no other statements – just colons) between the statements until the execution time for the line is exactly 1/50th (or 1/60th) of a second; that is, the two colours are stationary. (*Hint:* count a BORDER command as 10, a GO TO as 13 and a colon as 1; try to make the total value of the line, counted in this way, about 160.)

Complete Programs

I. Listing 13.1. Data required: 10 sets of X/Y coordinate pairs. The screen shows a zeroed table of these values, followed by "End.". A cursor, which points to the first row, can be moved down by Capital "6" and up by Capital "7". When you are at the required row, type "EDIT" (Capital "1") and the machine requests the pixel X/Y coordinates for that entry in the table. When you have finished move the cursor to "End.", then EDIT and type "y"(es). The program will then draw a polygon by joining the 10 coordinate points.

II. Listing 13.2. Data required: numeric key input for mythical file system. Type "1" to "6". For "5" or "6" the machine checks if you really mean it: type "y"(es) or "n"(o). BREAK terminates program.

III. Listing 13.3. No data required: BREAK terminates program.

IV. Listing 13.4. Type any key on request. No data required: BREAK terminates program.

V. Listing 13.5. After the BEEP, program requires five pictures to be loaded from tape. Type "1", "2", "3", "4" or "5" for instant display of required frame, 'm' for 'movie' and 's' to stop movie.

VI. Listing 13.6. When screen is full, hold down any key for scroll to continue. BREAK to stop.

VII. Listing 13.7. BASIC version of VI. Very slow!
VIII. Listing 13.8. No data required.
IX. Listings 13.7 and 13.9. Type RUN 9900 for Renumber and RUN 9970 for Delete. Example data: editing listing 13.7

```
RUN 9900
Renumber lines 10 to 30
Starting at 60 in steps of 5
```

Now LIST program to see changes

```
RUN 9970
Delete lines 60 to 99
```

Halt: there must be at least one line in front of section to be deleted

```
RUN 9970
Delete lines 65 to 99
EDIT (Capital "1")
LIST
```

14 A Worked Example of a Video Game

In this chapter we examine the limits of BASIC programming for animated video games. We have found that games written in BASIC are expensive, and in general the players' interest short-lived. If users can achieve a reasonable result themselves then they far prefer to write their own simple video games, and spend their money only on sophisticated games written in machine-code. Listing 14.1 is an example of the sort of game that most competent BASIC programmers can reasonably expect to write, without resorting to machine-code routines. The game, ISLAND DEFENCE (48K machines only), is a typical 'shoot-em up' game, but the techniques we discuss could just as easily be applied to 'bat and ball' (for example, TENNIS) or 'tactical' games (for example, PAC MAN).

Outline of the Game

On the screen we draw a scene of a green island with a small hill, dark blue sea, and a light blue sky containing a yellow sun and white cloud. There are three trees, and a light blue concrete area on which a tent is drawn. A man comes out of the tent and walks to a sandbagged gun emplacement, where he gives 20 bullets to the gunner and returns to the tent. Enemy aeroplanes appear out of the sun, fly over the hill and bomb the camp; the gunner defends the camp by firing at the aeroplanes. At every attack, the aeroplane is either disabled or makes a successful hit on the camp. With each successful hit the size of the tent diminishes. After 14 hits the tent disappears, and the next plane bombs the gun. On destruction of the gun the BORDER of the screen goes haywire and the scene is reset. After each bombing run, the plane flies through the cloud, over the gun and exits stage left. The stock of ammunition is replenished after 20 planes have completed their runs. After the gun has been bombed for the third time the screen goes blood red and the whole game starts again.

The gun has seven firing positions specified by keys "1" to "7", and a missile is fired by pressing "x". Because of speed restrictions only one plane and one missile can appear on the screen at any given time.

For you to make the most of the explanations in this chapter we advise you to get the companion cassette tape, and LOAD and RUN listing 14.1. The pro-

Listing 14.1

```
  9 REM initialise routines to allow use of alternate character set.
 10 CLEAR 62294: INK 0: PAPER 7: BORDER 7: FLASH 0
 20 DIM S(6): FOR I = 1 TO 6:  READ S(I): NEXT I
 30 DATA 15360,62039,62807,63575, 64343,64848
 40 LET set = 50: LET S = 1: GO SUB set: GO TO 200

 50 REM set/change to set S
 51 REM IN  : S
 60 LET HI = INT (S(S)/256): LET LO = S(S) - HI*256
 70 POKE 23606,LO: POKE 23607,HI: RETURN

 79 REM load in game characters for set 2.
 80 REM charload
 90 LET N$ = "gameset"
100 LET S = 2
110 INPUT ("    LOADING " + N$ + CHR$ 6 + "Start tape, then press enter.")
    ; LINE X$
120 LOAD N$ CODE (S(S) + 256),768: RETURN


200 REM main program
209 REM initialise variables pointing to routines.
210 LET charload = 80: LET load = 4600: LET create = 5000
    : LET credit = 5500: LET char = 5600
220 LET keyboard = 500: LET camp = 700: LET status = 800: LET plane = 980
230 LET reload = 1500: LET ammo = 1900: LET missile = 2200
    : LET explode = 2500
240 LET bombcamp = 3500: LET bombgun = 3000: LET hiscore = 5700: LET HSC = 0
249 REM if character set is not in place load it.
250 IF PEEK 62303 <> 110 THEN GO SUB charload
258 REM load background, create characters for fuel dump,
259 REM print names on bottom two lines.
260 GO SUB load: GO SUB create: GO SUB credit
269 REM make a copy of colour attributes for screen.
270 DIM A(704): FOR I = 1 TO 704: LET A(I) = PEEK (22527 + I): NEXT I
279 REM prepare strings for use by display routines.
280 LET S$ = "  SCORE      ": LET B$ = "  BASE      "
290 FOR I = 1 TO 4: LET S$ = S$ + CHR$ 8: LET B$ = B$ + CHR$ 8: NEXT I
300 DIM N$(9): LET N$=" ABCDEFG█"

309 REM start/restart for game.
310 BRIGHT 1: PAPER 8: INK 8: OVER 0
319 REM reset SCore and no. of BaSes, print out score line
320 LET SC = 0: LET BS = 3: GO SUB status
329 REM remove remaining ammunition and remove flash from gunners square.
330 PRINT AT 17,5;"  ": PRINT AT 18,5;"  ": PRINT AT 16,3;" "
339 REM restore colour attributes from copy.
340 FOR I = 1 TO 704: POKE 22527+I,A(I): NEXT I
349 REM print new tent and gunner, update score line, reload ammunition.
350 GO SUB camp: GO SUB status: GO SUB reload
358 REM prepare for main loop of program set 'p' to top of 'plane' cascade.
359 REM set 'm' to 'missile' ready state, set 'k' to keyboard routine.
360 LET p = plane: LET m = missile: LET k = keyboard
369 REM start a new wave of 20 AiRplanes.
370 LET AR = 20
379 REM start of main loop
380 GO SUB p: OVER 1: GO SUB m: OVER 0: GO SUB k
389 REM jump out of loop.
390 IF DEAD THEN GO TO 420
399 REM if more airplanes to come keep looping.
```

```
400 IF AR > 0 THEN GO TO 380
409 REM if end of planes then reload and start new wave, continue looping.
410 GO SUB reload: GO TO 370

419 REM make sure plane is gone, if you have any bases left use the next one.
420 INK 0: PRINT AT 12,0;"     ": IF BS > 1 THEN LET BS = BS - 1: GO TO 350

429 REM game over so flash border and turn screen blood red.
430 OVER 1: FOR I = 1 TO 22: BEEP 0.005,RND*30-15: BORDER RND*7
440 FOR J = 1 TO 2: PLOT 0,(22-I)*8: DRAW BRIGHT 1; PAPER 2; INK 2; 255,0
450 NEXT J: NEXT I: BORDER 7
460 OVER 0

469 REM check whether you beat the HiSCore then restart game.
470 IF SC > HSC THEN GO SUB hiscore
480 PAUSE 200: GO TO 310


500 REM keyboard
509 REM if no key pressed when routine checks then you missed your chance.
510 LET A$ = INKEY$: IF A$ = "" THEN RETURN
518 REM the fire key is pressed:-
519 REM if you're not out of missiles and a missile is ready then fire.
520 IF A$="X" OR A$="x" THEN IF m = missile AND NOT OUT THEN LET m = f
    : LET M$ = F$: GO SUB ammo: OVER 1: GO SUB m: OVER 0: GO TO 550
529 REM if the key is not '1' to '7' then wrong key press, ignore it.
530 IF A$ > "7" OR A$ < "1" THEN RETURN
538 REM calculate which routine is used if missile is fired in this direction.
539 REM set string which is used for missile and change gun direction.
540 LET f = 2000 + 10*VAL A$: LET F$ = CHR$ (83 + VAL A$)
550 PRINT AT 16,3;F$
560 RETURN

700 REM camp
709 REM print new gunner and tent, reset border to white.
710 PRINT AT 17,3;"z": BRIGHT 0: BORDER 7: PRINT AT 16,23;"vw"
    : PRINT AT 17,23;"xy": BRIGHT 1
718 REM set height of tent to 14, reset truth-flags,
719 REM use keyboard rotuine to initialise gun position.
720 LET YT = 14: LET HIT = 0: LET DEAD = 0: LET A$ = "3": GO TO 540

800 REM status
809 REM use set 1 to print out score and no. of bases left then back to set 2.
810 LET S = 1: GO SUB set
820 PRINT AT 21,0;S$;SC,B$;BS
830 LET S = 2: GO SUB set
840 RETURN

980 REM plane
989 REM ensure that 'bombgun' pointer always starts at top of cascade.
990 PRINT AT 13,5;" ": PRINT AT 15,4;" ": PRINT AT 17,3;"z"
    : LET bombgun = 3000
999 REM change pointer, if result is over 1000 then plane will be launched.
1000 LET p = 991 + INT (RND*11):RETURN
1009 REM launch aircraft, start pointer going down cascade, load bomb.
1010 LET AR = AR - 1: LET p = 1020: LET BOMB = 1: RETURN
1019 REM move plane one place to right.
1020 PRINT AT R,C;" IJ": LET C = C + 1: IF C < 10 THEN RETURN
1029 REM if plane is over 10 columns onto screen move pointer down.
1030 LET p =1040
1039 REM diagonal dive.
1040 PRINT AT R-1,C;" ": PRINT AT R,C;" N": LET R = R + 1: LET C = C + 1
    : PRINT AT R,C;"n": IF C < 20 THEN RETURN
```

```
1049 REM if over 20 columns across then move pointer, remove left over plane.
1050 LET p =1060: PRINT AT R-1,C;" "
1059 REM call the 'bombcamp' cascade each of the five times this section is used.
1060 GO SUB bombcamp: PRINT AT R,C;" IJ": LET C = C + 1: IF C < 25 THEN RETURN
1069 REM move pointer down and start curve up towards cloud.
1070 LET p =1080
1080 PRINT AT R,C;"  ": LET R = R - 1: LET C = C + 1: PRINT AT R,C;"KL"
     : IF C < 28 THEN RETURN
1090 LET p =1100: PRINT AT R,C;" ": LET C = C + 1
1100 PRINT AT R+1,C;" ": PRINT AT R,C;"o": LET R = R - 1
     : PRINT AT R,C;"O": IF R > 7 THEN RETURN
1110 LET p =1120: PRINT AT R+1,C;" "
1120 PRINT AT R,C;"  ": LET R = R - 1: LET C = C - 1: PRINT AT R,C;"kl"
     : IF R  > 3 THEN RETURN
1130 LET p =1140
1139 REM fly into cloud, plane still moves normally but you can't see it.
1140 PRINT AT R,C;"ij ": LET C = C - 1: IF C > 11 THEN RETURN
1150 LET p =1160: PRINT AT R,C;"   "
1159 REM start dive towards gunner.
1160 PRINT AT R,C;" ": LET R = R + 1: LET C = C - 1: PRINT AT R,C;"M"
     : PRINT AT R+1,C;"m": IF R < 6 THEN RETURN
1170 LET p =1180
1180 PRINT AT R,C;" ": LET R = R + 1: PRINT AT R,C;"P": PRINT AT R+1,C;"p"
     : IF R < 8 THEN RETURN
1189 REM third section of dive check and erase possible left over missile.
1190 LET p =1200: IF X <> 9 AND Y <> 7 THEN PRINT AT 7,9;" "
1200 PRINT AT R,C;" ": LET R = R + 1: LET C = C - 1: PRINT AT R,C;"M"
     : PRINT AT R+1,C;"m": IF R < 11 THEN RETURN
1210 LET p =1220: PRINT AT R,C;" ": LET R = R + 1
1219 REM last section of flight, call bombgun cascade each time.
1220 GO SUB bombgun: LET C = C - 1: PRINT AT R,C;"ij ": IF C > 0 THEN RETURN
1229 REM last two parts of cascade deal with plane going of screen.
1230 LET p =1240: PRINT AT R,C;"j ": RETURN
1239 REM reset plane's Row to 2 and set pointer back to top of cascade.
1240 PRINT AT R,C;" ": LET R = 2: LET p = plane
1250 RETURN


1500 REM reload
1509 REM if the camp is destroyed then dont reload ammunition
1510 IF HIT THEN RETURN
1519 REM explicit instructions for animation of figure.
1520 BRIGHT 0: LET R = 18
1529 REM walk from tent to trees.
1530 FOR C = 24 TO 19 STEP -1: PRINT AT R,C;"S": PRINT AT R+1,C;"s"
     : BEEP 0.01,-15: PAUSE 5
1540 PRINT AT R,C;"R": PRINT AT R+1,C;"r": PAUSE 3
1550 PRINT AT R,C;" ": PRINT AT R+1,C;" ": NEXT C
1559 REM use over-printing so that trees aren't damaged as figure gets close.
1560 OVER 1: PRINT AT R,18;"S": PRINT AT R+1,18;"s": BEEP 0.01,-15: PAUSE 5
     : PRINT AT R,18;"S": PRINT AT R+1,18;"s"
1570 PRINT AT R,18;"R": PRINT AT R+1,18;"r": PAUSE 3
     : PRINT AT R,18;"R": PRINT AT R+1,18;"r"
1579 REM make noises as though figure goes behind tree.
1580 FOR J = 1 TO 2: BEEP 0.01,-15: PAUSE 11: NEXT J
1589 REM show feet emerging from behind tree.
1590 PRINT AT R+1,15;"s": BEEP 0.01,-15: PAUSE 5: PRINT AT R+1,15;"s"
1600 PRINT AT R+1,15;"r": PAUSE 3: PRINT AT R+1,15;"r"
1609 REM show whole figure coming out of tree.
1610 PRINT AT R,14;"S": PRINT AT R+1,14;"s": BEEP 0.01,-15: PAUSE 5
     : PRINT AT R,14;"S": PRINT AT R+1,14;"s"
1620 PRINT AT R,14;"R": PRINT AT R+1,14;"r": PAUSE 3
     : PRINT AT R,14;"R": PRINT AT R+1,14;"r"
```

```
1629 REM walk from tree to end of strip, onto grass at column 8 with bright on.
1630 OVER 0: FOR C = 13 TO 8 STEP -1: BRIGHT (C = 8)
1640 PRINT AT R,C;"S": PRINT AT R+1,C;"s": BEEP 0.01,-15: PAUSE 5
1650 PRINT AT R,C;"R": PRINT AT R+1,C;"r": PAUSE 3
1660 PRINT AT R,C;" ": PRINT AT R+1,C;" ": NEXT C
1669 REM print figure at column 7 ready to refill ammo dump.
1670 PRINT AT R,C;"R": PRINT AT R+1,C;"r"
1679 REM refill ammo dump, with sound effects.
1680 FOR G = 6 TO 5 STEP - 1: FOR H = 18 TO 17 STEP - 1: FOR N = 2 TO 9
1690 PRINT AT H,G; INK 3;N$(N): BEEP 0.02,H+G-N
1700 NEXT N: NEXT H: NEXT G
1709 REM remove figure from column 7.
1710 PRINT AT R,C;" ": PRINT AT R+1,C;" "
1719 REM reset ammunition print and truth flags.
1720 LET N = 8: LET OUT = 0: LET H = 17: LET G = 6
1729 REM walk from end of strip to tree, bright off after column 8.
1730 FOR C = 8 TO 13: BRIGHT (C = 8)
1740 PRINT AT R,C;")": PRINT AT R+1,C;"+": BEEP 0.01,-15: PAUSE 5
1750 PRINT AT R,C;"(": PRINT AT R+1,C;"*": PAUSE 3
1760 PRINT AT R,C;" ": PRINT AT R+1,C;" ": NEXT C
1769 REM over-print as figure reaches tree.
1770 OVER 1: PRINT AT R,14;")": PRINT AT R+1,14;"+": BEEP 0.01,-15: PAUSE 5
     : PRINT AT R,14;")": PRINT AT R+1,14;"+"
1780 PRINT AT R,14;"(": PRINT AT R+1,14;"*": PAUSE 3
     : PRINT AT R,14;"(": PRINT AT R+1,14;"*"
1789 REM overprint legs as they go into tree.
1790 PRINT AT R+1,15;"+": BEEP 0.01,-15: PAUSE 5: PRINT AT R+1,15;"+"
1800 PRINT AT R+1,15;"*": PAUSE 3: PRINT AT R+1,15;"*"
1809 REM make noises as though figure is behind tree.
1810 FOR J = 1 TO 2: BEEP 0.01,-15: PAUSE 11: NEXT J
1819 REM overprint as figure emerges from tree.
1820 PRINT AT R,18;")": PRINT AT R+1,18;"+": BEEP 0.01,-15: PAUSE 5
     : PRINT AT R,18;")": PRINT AT R+1,18;"+"
1830 PRINT AT R,18;"(": PRINT AT R+1,18;"*": PAUSE 3
     : PRINT AT R,18;"(": PRINT AT R+1,18;"*"
1839 REM walk from tree to tent.
1840 OVER 0: FOR C = 19 TO 24: PRINT AT R,C;")": PRINT AT R+1,C;"+"
     : BEEP 0.01,-15: PAUSE 5
1850 PRINT AT R,C;"(": PRINT AT R+1,C;"*": PAUSE 3
1860 PRINT AT R,C;" ": PRINT AT R+1,C;" ": NEXT C
1869 REM reset Row and Column for use by plane routine.
1870 LET C = 0: LET R = 2: BRIGHT 1
1880 RETURN



1900 REM ammo/ print missile dump
1902 REM OUT : OUT
1909 REM take one of ammunition dump and check whether out of ammo.
1910 PRINT AT H,G;N$(N): LET N = N - 1
1920 IF N = 0 THEN LET N = 8: LET H = H + 1: IF H = 19 THEN LET H = 17
     : LET G = G - 1: IF G = 4 THEN LET OUT = 1
1930 RETURN

2000 REM missile directions
2009 REM remove old missile by overprinting and calculate new position.
2010 PRINT AT Y,X;M$: LET Y = Y - 3: LET X = X - 2: GO TO 2100
2020 PRINT AT Y,X;M$: LET Y = Y - 3: LET X = X - 1: GO TO 2100
2030 PRINT AT Y,X;M$: LET Y = Y - 3: GO TO 2100
2040 PRINT AT Y,X;M$: LET Y = Y - 3: LET X = X + 1: GO TO 2100
2050 PRINT AT Y,X;M$: LET Y = Y - 3: LET X = X + 2: GO TO 2100
2060 PRINT AT Y,X;M$: LET Y = Y - 2: LET X = X + 2: GO TO 2100
2070 PRINT AT Y,X;M$: LET Y = Y - 2: LET X = X + 3: GO TO 2100
```

```
2098 REM if missile is on same column as plane check for being near row of plane.
2099 REM if missile is close enough explode and reset missile to ready.
2100 IF X = C THEN IF ABS (Y - R) < 2 THEN GO SUB explode: LET m = missile
     : GO TO m

2109 REM if missile is off screen then reset pointer 'm' to ready.
2110 IF X < 0 OR Y < 0 THEN LET m = missile: GO TO m

2119 REM print missile at new position.
2120 PRINT AT Y,X;M$: RETURN

2200 REM missile/ready to fire
2210 LET X = 3: LET Y = 16
2220 RETURN

2500 REM explode
2508 REM missile has hit plane so blow up plane and add to score.
2509 REM check whether explosion has taken place in cloud or in sky.
2510 OVER 0: LET SKY = (R <> 3 OR C < 16)
     : IF NOT SKY THEN PRINT AT R,C-1;"   "
2519 REM central flash and low buzz, add to score for hitting plane.
2520 INK 2: IF SKY THEN PRINT AT R,C;"t"
2530 FOR I = 1 TO 5: BEEP 0.01,I - 10: BEEP 0.01,-I - 10: NEXT I: LET SC = SC + 1
2539 REM if explosion is seen then print cloud of debris.
2540 IF SKY THEN PRINT AT R-1,C-1;">@<": PRINT AT R,C-1;"$!&"
     : PRINT AT R+1,C-1;":%;"
2549 REM high pitched whizz.
2550 FOR I = -4 TO 4: BEEP 0.002,(50 + ABS I): NEXT I
2559 REM remove all trace of explosion.
2560 INK 0: IF SKY THEN PRINT AT R-1,C-1;"   ": PRINT AT R,C-1;"   "
     : PRINT AT R+1,C-1;"   "
2569 REM print out new score line, reset plane and missile pointers.
2570 INK 8: GO SUB status: LET m = missile: LET p = plane
2580 LET R = 2: LET C = 0
2590 RETURN


3000 REM bomb gun
3009 REM if camp isn't destroyed or plane hasn't got a bomb left then don't drop.
3010 IF NOT HIT OR NOT BOMB THEN RETURN
3019 REM place bomb on screen and start pointer down cascade.
3020 PRINT AT 13,5;".": LET bombgun = 3030: RETURN
3030 PRINT AT 13,5;" ": PRINT AT 15,4;".": LET bombgun = 3040: RETURN
3040 PRINT AT 15,4;" ": PRINT AT 17,3;".": LET bombgun = 3050: RETURN
3050 PRINT AT 17,3; FLASH 1;" ": LET bombgun = 3060: RETURN
3059 REM clear plane away, make sure any missiles shoot off the screen.
3060 PRINT AT 12,0;"      ": OVER 1: FOR I = 1 TO 7: GO SUB m : NEXT I
3069 REM flash border and make exploding noises.
3070 FOR I = 1 TO 50: BORDER RND*7: BEEP 0.01,RND*30 - 15: NEXT I
3078 REM set flag for end of gunner.
3079 REM set ink to sky-blue  subsequent movement of plane is invisible.
3080 LET DEAD = 1: OVER 0: INK 5
3089 REM reset cascade.
3090 LET bombgun = 3000: RETURN

3500 REM bomb camp
3509 REM if the camp has gone save bomb for the gunner.
3510 IF HIT THEN RETURN
3519 REM mark bomb as dropped and start cascade.
3520 LET BOMB = 0: PRINT AT 14,21;".": LET bombcamp = 3530: RETURN
3530 PRINT AT 14,21;" ": PRINT AT 15,22; BRIGHT 0;".": LET bombcamp = 3540
     : RETURN
```

```
3540 BRIGHT 0: PRINT AT 15,22;" "
3549 REM explode tent.
3550 PRINT AT 16,23; OVER 1; INK 2;"#$"
3560 PRINT AT 17,23; OVER 1; INK 2;"%&"
3570 LET bombcamp = 3580: BRIGHT 1: RETURN
3579 REM remove explosion from tent.
3580 BRIGHT 0: PRINT AT 16,23; OVER 1; INK 0;"#$"
3590 PRINT AT 17,23; OVER 1; INK 0;"%&"
3600 LET bombcamp = 3610: BRIGHT 1: RETURN
3609 REM remove top remaining line of tent.
3610 PLOT INVERSE 1;184,33 + YT: DRAW INVERSE 1;16,0
3619 REM check whether tent is gone and reset bombcamp cascade.
3620 LET bombcamp = 3500: LET YT = YT -1: LET HIT = (YT = 0)
3630 RETURN

4600 REM load
4609 REM load picture from tape.
4610 LET N$ = "background"
4620 INPUT ("    LOADING " + N$ + CHR$ 6 + "Start tape, then press enter.")
     ; LINE X$
4630 LOAD (N$)SCREEN$
4640 RETURN

5000 REM create/characters
5009 REM make characters for ammo dump in user defined graphics set.
5010 LET D = 255
5020 FOR I = 0 TO 6: FOR J = 0 TO 7
5030 LET P = USR "G" - I*8 + J
5040 IF J > I POKE P,D
5070 NEXT J: NEXT I
5080 RETURN

5500 REM credit
5509 REM print on bottom two lines.
5510 DEF FN S(R,C) = 16384 + INT (R/8)*2048 + (R - INT(R/8)*8)*32 + C
5520 DEF FN C(A$) = 15360 + CODE A$*8
5530 LET R = 22: LET A$ = "]/ISLAND DEFENCE/[  BY  BJJ & IOA"
5540 FOR J = 1 TO LEN A$: LET C = J - 1: GOSUB char: NEXT J
5550 LET R = 23: LET A$ = "  HI-SCORE       SCORED BY BJJ  "
5560 FOR J = 1 TO LEN A$: LET C = J - 1: GOSUB char: NEXT J
5570 RETURN

5600 REM char
5609 REM print j'th character of a$ at r,c,
     change attribute to red paper and white ink, beep.
5610 LET AT = FN S(R,C): LET FROM = FN C(A$(J))
5620 FOR I = 0 TO 7: POKE AT + I*256, PEEK (FROM + I): NEXT I
5630 POKE 22528 + C + R*32,87: BEEP 0.03, CODE A$(J)-50
5640 RETURN

5700 REM hiscore
5709 REM change hi-score and print it on bottom of screen.
5710 LET HSC = SC: LET A$ = STR$ HSC
5720 LET R = 23: FOR J = 1 TO LEN A$:  LET C = 10 + J: GOSUB char: NEXT J
5729 REM change name to flashing letters.
5730 FOR I = 23291 TO 23293: POKE I,PEEK I + 128: NEXT I
5739 REM get three initials for hiscore from key board.
5740 LET J = 1: FOR C = 27 TO 29
5750 IF INKEY$ <> "" THEN GO TO 5750
5760 IF INKEY$ = "" THEN GO TO 5760
5770 LET A$ = INKEY$: GO SUB char: NEXT C
5780 RETURN
```

gram will ask you to LOAD first the special character set, 'gameset' (for drawing the plane, tent, man, etc.), and then the 'background' (which sets the scene), before starting the action. The listing is well documented, so there is no need for us to go into too much detail here. We shall simply outline a general approach for constructing these games, and describe methods of solving typical problems that arise.

The Foreground

It is essential to carefully plan your game before you start writing the program. You must first draw a rough plan of the proposed game scene on graph paper. Then sketch in the positions of fixed objects (for example, the sun and clouds) and also the areas to be traversed by moving objects (for example, the path of character blocks that at some time will contain characters to make up the planes). This should fix the scale of the objects to be used, and will give a general impression of the final game. Time spent in careful planning at this stage will be a fraction of that needed to adjust a nearly completed program. You must ensure that the proposed display can actually fit into the graphics area, and that you never get a situation where a moving object introduces a third colour within a character block. Once you have decided on a screen layout, you must then create the objects for the foreground (for example, explosions, the plane, the man) – that is, the moving parts. The fastest BASIC command for putting a large object on the screen is the PRINT statement. So we build our objects out of defined character blocks, and use the CHARACTER GENERATOR program of chapter 5 to produce the required shapes. To get exactly the shape you want for a display, you should repeatedly examine the characters at their normal size and re-edit them if you are not satisfied. It may be helpful to add parts of your game-program at the end of the CHARACTER GENERATOR program (option 7, see chapter 5). This will allow you to observe these objects in action while their final form is still under consideration.

Exercise 14.1
Add an extra option to the CHARACTER GENERATOR program so that you can edit a single character from within a group. This option should POKE the eight BINary values of the character grid being edited, together with the other characters in the group, into the display-file locations for specified positions on the screen; for example, the two blocks that make up the plane in horizontal flight to the right (see figure 14.1). This quickly allows you to evaluate the merits of altering one pixel within a character.

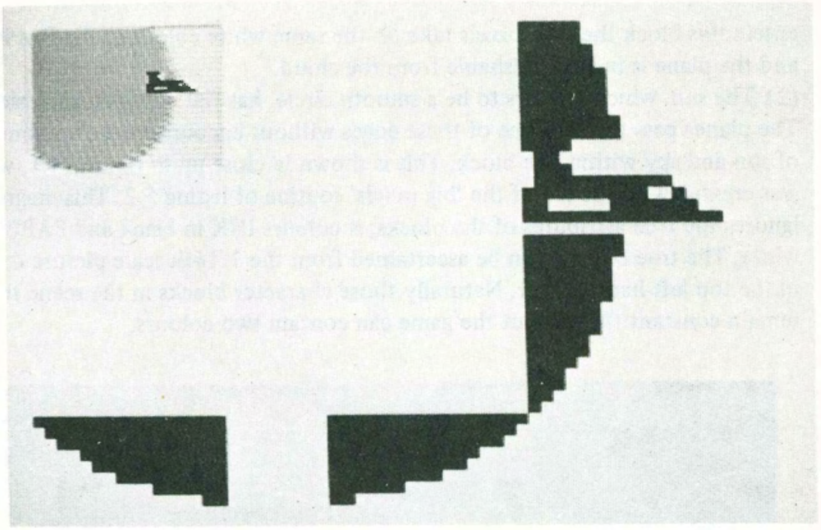


Figure 14.1

The Background

The background is now created using a combination of the CHARACTER GENERATOR and the diagram-construction routines from chapter 6. You first add large character blocks of colour using 'paper' for rectangular areas that approximate to large parts of the scene (for example, grass, sky, sun, etc.). Then superimpose the initial detail by PRINTing special characters (for example, the trees, sandbags, and the edges of the sun, grass and clouds etc.). This can be given final touches with the 'point' and 'line' options of the diagram routines. For example, the trees are made by adding lines and dots to the foreground explosion characters. We can expand the range of colours by allowing BRIGHT as well as normal colours. Special care must be taken when constructing blocks through which moving objects will pass. They must contain only one background colour with no detail. Remember that only two colours are allowed in any one block, and there is always a danger that a moving object could introduce a third colour. In our background (see figure 14.2) there are two particular places where care is needed.

(1) The planes pass over and behind a cloud on their return run, so the cloud contains a path of white blocks through its middle. Note that blue sky and white cloud cannot occur within the same character block on the path of the plane, so the boundary between sky and cloud on this path must be a straight edge. The plane actually disappears behind the cloud during its flight, so some of the blocks in this path contain white INK and white PAPER. Thus when the plane

enters this block the INK pixels take on the same white colour as the PAPER and the plane is indistinguishable from the cloud.
(2) The sun, which appears to be a smooth circle, has flat edges on each side. The planes pass through one of these edges without encountering a combination of sun and sky within one block. This is shown in close-up in figure 14.1, which was created with the aid of the 'big pixels' routine of listing 5.2. This diagram ignores the true attributes of the blocks; it colours INK in black and PAPER in white. The true colours can be ascertained from the 1/16th scale picture drawn in the top left-hand corner. Naturally those character blocks in the scene that remain constant throughout the game can contain two colours.

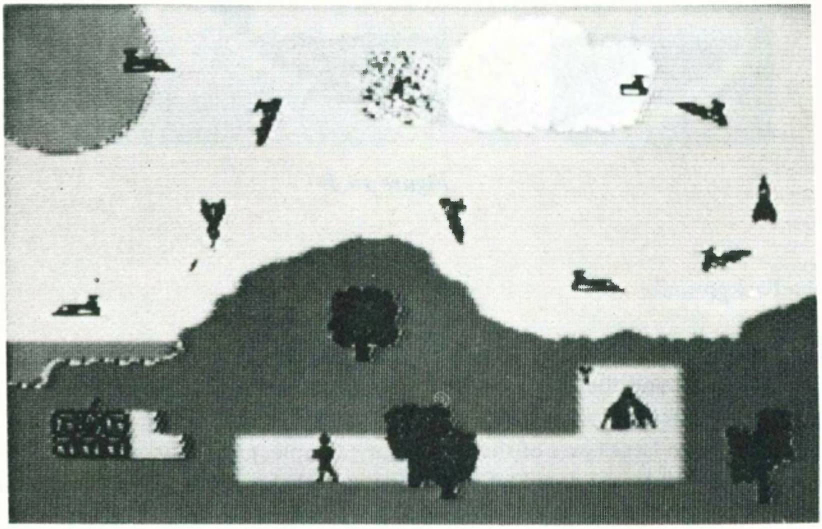


Figure 14.2

Cascade Animation

Once we have stored this first approximation of the background on tape, we need to write the routines that will position and move the foreground objects about the screen. Since we are aiming at speed of execution we must make these routines as explicit as possible. We need to minimise the amount of calculation required while the game is being played. Any extra programming or calculation, however long, which, if done prior to the play would save precious milliseconds during the play, should be implemented. We describe one such technique in our example program, the popular *cascade*, or *variable entry point* method. The timewise-important routines in the program are referred to by a variable pointer (an identifier given in lower case). These routines are made up of a chain (or cascade) of explicitly programmed subtasks, only one of which will be undertaken on each separate entry to the routine. The solution to each subtask is laid out as a set of lines, after which the pointer to that routine is altered and we return to the calling program. On re-entry, the program naturally obeys a different section of code within that routine, usually the pointer changes to the next subtask down the cascade. With each call, this process continues with the pointer moving down through the sections of the cascade until it eventually reaches the bottom (where it will usually be reset to the top). The complete game will consist of an initialisation phase followed by a loop of calls to routines, each of which is a cascade that solves a specific problem within the game (for example, to move the aeroplane or to bomb the base). Cascade routines may even call other cascades! This gives an impression of a parallel process, which is essential if users are to believe that they are playing a game in which a number of different events are happening simultaneously. In our game we try to give the impression that the plane, gun and missiles can all move independently.

Our game is a little too complicated to be an elementary illustration of this technique, so we introduce another, very much simpler, game (available on 16K machines as well). Consider the following two programs that perform independent functions. Listing 14.2 waits until a key is pressed then shoots a dot across the screen; listing 14.3 continuously moves a plus sign up the screen in a zig-zag pattern. Both programs use the fast animation techniques found in the ISLAND DEFENCE program; however, because each is so small, it is necessary to slow them down with a PAUSE command (note that excess speed is only a problem when programs are very simple). From these listings we create two cascade routines (listing 14.4) so that the dot and cross appear to move simultaneously.

The dot cascade is in three parts: (a) place a dot at row 11, column 0; (b) if the user hits the keyboard then start the dot moving; and (c) move the dot one column farther across the screen until it hits the right-hand edge. Whenever one of these processes is finished the identifier of the routine 'dot' is moved down to the next section. After the dot has moved right across the screen, 'dot' is reset to the start position.

The cross cascade is also in three parts: (a) place the cross at row 22, column 8; (b) move the cross one row up and diagonally to the right on the bottom half of the screen; and (c) move one row up and diagonally to the left in the top half, and if the dot and cross lie in the same block, then SPLAT.

The two routines are turned into a game by a main routine that loops repeatedly through calls to 'dot' and 'cross', where, of course, these identifiers are perpetually changing so that they each refer to the correct part of their cascade.

Listing 14.2

```
200 REM dot
210 PRINT AT 11,0;"."
220 IF INKEY$ = "" THEN GO TO 220
230 LET D = 0
240 PRINT AT 11,D;" ":LET D = D + 1
250 IF D = 32 THEN GO TO 210
260 PRINT AT 11,D;"."
270 PAUSE 1: GO TO 240
```

Listing 14.3

```
300 REM cross
310 LET R = 21: LET C = 8
320 PRINT AT R,C;"+"
330 PRINT AT R,C;" ":LET R = R - 1
340 IF R < 0 THEN GO TO 310
350 IF R > 11 THEN LET C = C + 1
360 IF R <= 11 THEN LET C = C - 1
370 PRINT AT R,C;"+"
380 PAUSE 1: GO TO 330
```

Listing 14.4

```
100 REM main loop
110 LET dot = 200: LET cross = 300
120 GO SUB dot: GO SUB cross
130 GO TO 120


200 REM dot cascade
210 PRINT AT 11,0;".": LET dot = 220: RETURN

220 IF INKEY$ = "" THEN RETURN
230 LET D = 0: LET dot = 240: RETURN

240 PRINT AT 11,D;" ":LET D = D + 1
250 IF D = 32 THEN LET dot = 210: RETURN
260 PRINT AT 11,D;"."
270 RETURN


300 REM cross cascade
310 LET R = 21: LET C = 8
320 PRINT AT R,C;"+": LET cross = 330: RETURN

330 PRINT AT R,C;" ":LET R = R - 1
340 LET C = C + 1
350 PRINT AT R,C;"+"
360 IF R > 11 THEN RETURN
370 LET cross = 380: RETURN

380 IF R = 11 AND C = D THEN PRINT AT R,C;"SPLAT": STOP
390 PRINT AT R,C;" ":LET R = R - 1
400 IF R < 0 THEN LET cross = 310: RETURN
410 LET C = C - 1
420 PRINT AT R,C;"+"
430 RETURN
```

Exercise 14.2

Add a line to the calling loop of the above program so that after a SPLAT the pointers to the top of the cascades are reset and the score (number of hits) printed.

Write a 'duck shoot' game where a hunter, under keyboard control, moves left and right at the bottom of the screen. He shoots at ducks that fly left to right, up and down, across the screen.

Further Animation Techniques

With such a simple game we find that the saving, in time and programming effort, from using the cascade technique is very small. However in larger programs, which can have complicated cascade programs (for example, the movement of the plane in ISLAND DEFENCE), the time savings can make the difference between a good fast game and a boring slow one. Note that there are eight different sets of character blocks that describe the plane; see figure 14.2, which shows the complete scene and one example of each plane, as well as other foreground objects. The cascade for drawing the plane is divided into sections, each of which places one type of plane in a particular area of the screen.

In this game we show a variety of different ways of solving the problems of animation. The planes are moved by printing them in transparent INK, and then obliterated from their previous positions with blanks. The missile characters are OVERprinted (again with transparent INK) on the existing detail and then erased at their last position by OVERprinting with the same character. These methods have both advantages and disadvantages. Where removal of the old position can be combined with the printing at the new (see the sections where the plane flies horizontally) you will find that normal PRINTing often takes less time, but it does mean you lose any detail in the background. OVERprinting takes longer, but leaves detail undamaged. Using a combination of these techniques in a program can cause problems if two moving objects pass simultaneously through the same block: for example, what happens if a square containing a missile is blanked out by a passing plane? This can occur in the game at character block row 7, column 9. The effect is that the OVERprinted missile, which is supposed to cancel out the old one, is left hanging in mid-air. Discretion is the better part of valour and it is far better (timewise) to check for these problems and explicitly program a cover-up, rather than add complexity into your algorithm and perhaps even introduce another fault. The plane cascade contains an extra statement (at line 1190), which ensures that any such mishap is swiftly covered. When trying to remove faults, remember that a brute-force cover-up will probably be far quicker (in your time, and running time) than a fancy fault-avoidance routine. Although you should have foreseen most problems in the planning stage, and adjusted the background and the paths of moving objects, some peculiarities are certain to occur.

There are two places where we have shown an object apparently passing behind a background object. In order that a plane can vanish behind the cloud, we simply made the INK white in those blocks where the plane was to disappear. Since a white plane in a white cloud is about as easy to spot as a black cat in a coal-cellar at midnight, we are tricked into thinking that the plane is hidden by the cloud. For the man marching behind the tree, the whole problem becomes more complicated. We still wish to see the tree in two colours when he is behind it. The trick this time is to make the man walk up to the tree normally; OVERprint him when he is in the same block as outer foliage, and then simply make marching noises, without printing, for the appropriate length of time, before starting him off again from the other side of the tree. His legs must also reappear from behind the tree one block before his head when he is moving right, and equivalently when moving left. All these details must be carefully calculated before even writing the first line of BASIC code.

A combination of these techniques for moving objects, making an allowance for them to pass each other, will enable you to produce displays of very high quality. Of course you must use machine-code routines if you require really fast and complex games. Even so, many of the routines can still be in BASIC. There is no need to produce programs written completely in machine-code unless you want to sell your games.

Finally, as your games programs become more and more interwoven and cross-connected, you must keep a simple overview of the program. Note what tasks need to be done at the top level, use sensible variable names, put in plenty of comments during development, and above all, don't panic!

We leave you now with your Spectrum. It has proved reliable and sturdy, straightforward and easy to use. We are certain that you will have many, many hours (years!) of pleasure out of this machine. To start you off we give a number of ideas for projects in the next section. Good luck and good programming.

Complete Programs

I. Listing 14.1: the ISLAND DEFENCE game. We do recommend that you obtain the tape because you will find it time-consuming setting up the special character set and background. However, you should at some time create such character sets and backgrounds yourself.
II. Listing 14.2 ('dot'). Type any key.
III. Listing 14.3 ('cross'): no data required.
IV. Listing 14.4 ('main program', 'dot cascade' and 'cross cascade'). Type any key.

15 Projects

I. Use your Spectrum to draw a digital clock. Use the special large characters for the digits and a colon to separate them. Your clock can be made to keep correct time by using the internal clock of the Spectrum (see page 130 of the Spectrum BASIC Handbook (Vickers, 1982)).

II. Make a program that tests the Morse Code proficiency of the user. The content for the program should be a paragraph of text; after translating into Morse, the Spectrum should print out the dots and dashes using the medium-resolution character blocks. It should also use BEEP to simulate the sound of Morse. Your program should have a variable rate of production of the Morse Code, so that the speed of the test can increase as the user becomes more proficient.

III. Draw a set of international road signs. Your program should draw the background of the figures (for example, red triangles); then use your own special routines or the programs of chapters 5 and 6 to finish off the foreground.

IV. Construct crossword puzzles on your television set. Each square of the puzzle should be 2 by 2 character blocks. The four blocks can be either black (in which case nothing goes in the square), or white, with the bottom left-hand corner holding the letter of a solution and the top two characters the clue number (if any). This allows space for a 16 by 11 puzzle.

As you have also to place the clues on the screen, there will obviously be a shortage of space. This problem can be solved by using the ideas of the 'slide show' of chapter 13; put the puzzle in one frame, and the clues on the remaining four frames. Solutions to the puzzle can be added by a 'cursor' method or by having a special input code; for example, letter "A" (across) or "D" (down) followed by the number of the clue, followed by your solution. If you make the puzzle smaller you could even have the option of using this code to bring clues on to the screen one at a time (or perhaps place them in rows 23 and 24), in which case the crossword need not be moved off the screen.

V. The Spectrum BASIC Handbook (Vickers, 1982) gives a program to draw the Union flag. Write programs or use the character generator and the diagram routines (chapters 5 and 6) to produce other flags. You can draw company logos, or even design new ones. Use the techniques in chapter 13 for accessing display-file locations to add an extra option to the diagram routines. This option allows you to copy a set of character blocks (already on the screen and specified by 'cursor') on to another set of blocks of the same size elsewhere on the screen (also specified by 'cursor'). You could even rotate or reflect them!

VI. The Spectrum BASIC Handbook (Vickers, 1982) shows how to use BEEP to create music(?). While BEEP is making the sounds, you can draw the musical notation on the screen. Construct the staves and then use special characters to place quavers, minims, etc. on the screen. The old music hall method of the 'bouncing ball' could be used to beat the time of the tune.

VII. Use the character block method to draw mazes. Naturally your program must generate mazes with real solutions. Give yourself time limits for getting through the maze. You can make the mazes dynamic, so that they change as the game progresses. Add extra problems: man-eating monsters that roam the maze; holes that suddenly appear and can swallow you up: 'space warps' that can transfer you anywhere in the maze if you do not move fast enough.

VIII. Extend the ideas of chapter 6. Draw your own special histograms, pie-charts and graphs. Make them dynamic (either by the 'movie' method of chapter 13, or the 'worm game' method of chapter 1, and its extended form in chapter 14). Generate whole sets of special characters. Create (apparent) three-dimensional histograms by drawing every bar in three different-coloured sections. The height of the front section of each bar must be a whole number of character blocks and two character blocks wide. A side section is one block wide, the same height as the first section, with a triangular hat filling the bottom-right half of a block. The third section lies above the first section and is in the shape of a rhombus that touches the first two sections. This ensures that there are never more than two colours in any character block, and makes the bar look like an orthographic view of a rectangular block.

IX. Create patterns. Use OVER with large numbers of random lines about the screen. Or draw lines in a dense but regular way to get Moiré patterns. For example, draw lines joining the points (0, I) to (255, 175 – I) for $0 \leqslant I \leqslant 175$, and points (I, 0) to (255 – I, 175) for $0 \leqslant I \leqslant 255$. Extend the ideas of the pattern program of chapter 5 to produce complex symmetrical patterns – any introductory book on crystallography (for example, see Phillips, 1956) will give you lots of ideas.

X. The crystallography books (for example, Phillips, 1956) will give you many ideas for three-dimensional objects. Extend into four dimensions – vertices are simply a vector of four numbers and they require 5×5 matrices for transformations. For an orthographic projection of a four-dimensional point, we simply ignore two of the coordinates (as opposed to one, z, in three dimensions). What are translation, scale and rotation in four dimensions?

XI. We have already presented two board games, Chess and Master Mind. There are many more possibilities: draughts (or checkers), Scrabble, Hangman, ludo. You can create a compendium of games. The Spectrum can simply act as the board, or it can also be a referee. If you feel really adventurous it can even act as an opponent.

XII. Use special characters to construct a deck of playing cards. These can be incorporated into a program to play blackjack (or pontoon) with the Spectrum acting as the bank and opponent.

XIII. You can draw certain types of brain-teasers on your television. For example, suppose you have nine squares and each is divided into quarters down the diagonals. Each quarter has a colour (blue "1", red "2", magenta "3" and green "4") and a sign ("+" or "–"). We represent each square as a sequence of four numbers that denote the areas taken clockwise around the centre. For example, we could have (–1, –2, 1, 4), (–1, 3, 4, –2), (1, –4, –2, 3), (1, 2, –3, –4), (1, 3, –2, –4), (1, 4, –3, –2), (2, –3, –4, 3) and two occurrences of (–1, –4, 3, 2). The problem is to place the nine squares in a three-by-three arrangement, so that if two quarters on neighbouring squares touch, then they must be of the same colour but of opposite sign. You can use the Spectrum to draw the squares initially on the left side of the screen and a three-by-three grid of the same sizes on the right. Then you take squares from the left and place them in the grid, or replace them back to the left.

Write a program to find a solution of the above problem – it takes about 10 minutes to run, and finds two independent solutions.

XIV. Produce a medium-resolution graphics package for manipulating 2 × 2 quarter blocks. This package should be similar to the one we gave for character blocks in chapter 5. The screen thus consists of 64 by 44 quarter blocks. Take a photograph of yourself and superimpose a grid of 64 by 44 on it. For each square, decide whether it is mainly light or dark, and colour the corresponding quarter block accordingly. This seems like a lot of work; but note that most of the picture will be a light background, so if we use white PAPER and black INK most of the squares need not be considered. Since a head measures more in height than it does in breadth, you can get greater resolution if you draw the head sideways on the screen. You could draw two heads side by side on the screen.

XV. Write a PAC MAN type of video game. This involves drawing five moving objects on the screen at a time. In order to make the game move faster, allow only two of the ghosts to move with each move of PAC MAN. The ghosts should find the shortest path towards the player when they are in hunting mode, and the best escape route in running mode. Because of the complex layout of the screen, you will have to compromise. Simply move towards (or away from) the player if there is no wall in the way. Find a quick way of coding their movements as this will be the most time-consuming part of the game. Speed is the essence of a good video game. Perhaps a simple machine-code routine could be used to print all five figures on the screen after altering their 'PRINT AT' positions.

XVI. Write a program that first (OVER)prints a graphics menu of special symbols on the left-hand side of the screen, for example, the stylised components for electronic circuits (resistors, capacitors, etc.). These symbols should consist of groups of character blocks. Use a cursor to point at any menu-symbol and then (using OVER) drag a copy of it to a required position on the screen. Also add a facility for drawing connecting lines and labelling with thin numeric and special characters (for example, Ω for ohms). You should also allow deletion of symbols inadvertently placed in the wrong position. Extra options could include saving and loading, as well as deleting the menu from the final diagram.

XVII. In all our perspective diagrams it is assumed that the objects lie totally in front of the eye. Change our programs so that they deal with the general case where vertices may be behind the eye. See Newman and Sproull (1973) concerning this three-dimensional clipping.

Appendix A Implementing Programs on the 16K Spectrum

Over 7K of this machine is used for the display and attribute files and for system variables. This automatically limits the size of programs and data to about 8.8K. Many of the programs given in this book far exceed this value, although most will run if the reader obeys the following proposals.

(1) Delete all REMarks from the tape listings and any unused routines from library files (for example, 'plot' and possibly 'scale') before MERG(E)ing the routines. Also follow the hints given in chapter 13 for optimising the program code. For example, the program for drawing the jet (listings 'lib1', 'lib3' and '9.9') will just fit into the store if the REMs, 'scale' and 'plot' are deleted.
(2) It is possible that there will not be enough store for the four alternative character sets (sets 2 to 5) and the User-Defined Graphics set (set 6). If there is enough store the address table for the six sets (set 1 is the standard set) should hold the values 15360, 29271, 30039, 30807, 31575 and 32080 respectively. It is possible that the use of sets with the lower addresses could corrupt your program and data, or even *crash* the computer. Use only the sets that do not interfere with the store! The CLEAR statement found at the beginning of programs that use alternate characters must be changed from CLEAR 62294 to CLEAR 255 + the address of the lowest set available (greater than 1). You will find that the CHARACTER GENERATOR program does not have enough space for set 2, and in this case we must CLEAR 255 + 30039. If you have a program that requires set 2 then create these characters as set 5 (say), store it on tape, and load it into your program, which must, of course, have space for set 2.
(3) You can break down programs into non-interdependent parts and store the pictures and/or data produced by them on tape. These files may be reloaded in conjunction with the other programs for further manipulation; for example, the diagram constructions of chapter 6. You should LOAD 'libdiag' and MERGE one of the histogram, pie-chart, graph or picture-editing programs. When you run the program you must remember not to attempt to use routines that are not currently in memory. You can SAVE and LOAD the intermediate pictures using the SCREEN$ option, and arrays using the DATA option. You can then LOAD the picture-editing routines to finish off your diagrams.
(4) Unfortunately, some programs will not fit into 8.8K of memory whatever is

done; these are the general hidden line algorithm for non-trivial objects, the movie program and the ISLAND DEFENCE game. If you are serious about studying computer graphics on the Spectrum we would strongly advise you to buy the 32K expansion for your machine.

Appendix B BASIC Program Listings

We now give a list of the BASIC program listings stored on the companion audio-cassette tape. These routines are in the form necessary for running on the 48K version of the Spectrum. Most of them need no changes in order to run on a 16K machine. However, if you have this type of machine you should check the REMarks in listings given in the book for any changes, and read appendix A. 'movie', the five frames 'sphere1' to 'sphere5' and ISLAND DEFENCE are the only routines that are 48K specific. You will find that with chapter 6 you have to LOAD 'libdiag' and MERGE just one of '6.8', '6.9', '6.10&11' and '6.12' in order for it to fit in the store. Should you need more than one type of data graph on the screen at any one time, you must intermediately SAVE picture on tape and reLOAD the screen after a new program has been created.

SIDE 1

File Name	*Contents*
directory 1	
lib1	Listings 2.1, 2.2, 2.3, 2.4, 2.8 and 3.3; routines for mapping two-dimensional Euclidean space on to graphics area.
2.9	Listing 2.9; joining points of regular N-gon.
2.12	Example of envelope.
2.13	Spirograph.
3.1	Square inside square inside square etc.
lib2	Listings 3.4, 4.1a, 4.2a, 4.3a, 4.4a, 4.5, 4.6; routines for matrix manipulation of two-dimensional space.
4.10	Construction of a general ellipse.
4.11 } 4.12 }	Construction and view of four 'space ships'.
7.1	Point of intersection of line and plane in three-dimensional space.
7.2	Point of intersection of two lines in three-dimensional space.
7.3&4	Example of dot and vector product.
7.5	Inverse of a 3 X 3 matrix.
7.6	Point of intersection of three planes in three-dimensional space.
7.7	Line of intersection of two planes in three-dimensional space.
lib3	Listings 3.4, 9.1, 9.2 and efficient rewrites of 8.1, 8.2, 8.3 and 8.4; routines for matrix manipulation of three-dimensional space.

8.5&6	Rotation of point about a general axis of rotation.
9.6&7&8	scene3, construction and drawit routines; necessary for drawing an orthographic view of two cubes.
9.9	Additional routines for drawing the jet (orthographic).
9.10&11	Routines for constructing an orthographic body of revolution.
10.1	Orientation of a three-dimensional triangle.
9.3&10.2	scene3 and hidden surface routine for drawing a cube.
10.3	Hidden surface routine for drawing a convex body of revolution.
0.1	Construction routine for a flying saucer(!).
10.4	Hidden surface routine for drawing a mathematical surface.
11.1&2	Perspective view of two cubes.
12.1	General hidden line algorithm.
12.2	Hidden line perspective view of two cubes.
12.3&4	Construction routines for cuboctahedron and icosahedron.
12.5&6&7	Construction routines for two stars, and a scene3 routine.
movie	Program for displaying five frames in quick succession.
sphere1 sphere2 sphere3 sphere4 sphere5	Five frames needed to make a movie of a rotating sphere.

SIDE 2

File Name	*Contents*
directory 2	
delete renumber	Utilities from listing 13.9.
5.5	The CHARACTER GENERATOR.
5.4	Main program for simple tessellated patterns.
libdiag	Listings 6.1, 6.4, 6.6; support routines for diagram construction.
6.2	'paper' and 'ink'.
6.3	'point' and 'line'.
6.5&7	'label', 'thin' and 'create'
6.8	Histogram routine/type 1
6.9	Histogram routine/type 2
6.10&11	Pie-chart and hatching routines.
6.12	Scientific graph routine.
thin3	Characters stored in set 3 and used to make thin characters.
thin4	Characters stored in set 4 and used to make thin characters.
1.16	The WORM GAME.
5.6	MASTER MIND program.
masterset	Characters needed by MASTER MIND program.
5.7	The Chess board program.

chesspiece	Character set holding the chess pieces.
14.1	ISLAND DEFENCE game.
gameset	Character set needed for ISLAND DEFENCE.
background	Scenery for the game.
1.4	Fractal program.
1.5	OVER patterns.
5.1	Displaying the binary construction of a given chracter.
5.2	The big pixels program.
5.3	Constructing a graphics character from eight binary numbers.
13.1	Menu input for 10 vertices of a polygon.
13.6	Scrolling of screen with wrap around: in machine-code.
13.7	Same as 13.6, but written in BASIC.
13.8	Program that lists itself, giving all numeric codes.
14.4	Simple cascade program: SPLAT.

References and Further Reading

References

Ahl, D. H. (Ed.) (1980). *Basic Computer Games*. Workman Publishing Co., New York

Bain, G. (1972). *Celtic Art: The Methods of Construction*, 2nd edition. Maclellan, Glasgow

Cohn, P. M. (1961). *Solid Geometry*. Routledge and Kegan Paul, London

Coxeter, H. S. M. (1974). *Regular Polytopes*. Dover Publications, New York

Davenport, H. (1952). *The Higher Arithmetic*. Hutchinson, London

Finkbeiner, D. T. (1978). *Introduction to Matrices and Linear Transformations*, 3rd edition. W. H. Freeman, San Francisco

Horowitz, E. and Sahni, S. (1976). *Fundamentals of Data Structures*. Pitman, London

Hurley, R. (1982). *More Real Applications for the ZX81 and the ZX Spectrum*. Macmillan, London

Hutty, R. (1981). *Z80 Assembly Language Programming for Students*, 2nd edition. Macmillan, London

Knuth, D. (1972, 1973, 1981). *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, 2nd edition, 1973. *Volume 2: Semi-numerical Algorithms*, 2nd edition, 1981. *Volume 3: Sorting and Searching*, 1972. Addison-Wesley, London

Liffick, B. W. (1979). *The BYTE Book of Pascal*. Byte Publications, New Hampshire

Mandelbrot, B. B. (1977). *Fractals*. W. H. Freeman, San Francisco

McCrae, W. H. (1953). *Analytical Geometry of Three Dimensions*. Oliver and Boyd, London

Newman, W. M. and Sproull, R. F. (1973). *Principles of Interactive Computer Graphics*. McGraw-Hill, London

Phillips, F. C. (1956). *An Introduction to Crystallography*, 2nd edition. Longmans, London

Stroud, K. A. (1982). *Engineering Mathematics*, 2nd edition. Macmillan, London

Tolansky, S. (1964). *Optical Illusions*, Pergamon, New York

Vickers, S. (1982). *Sinclair ZX Spectrum: Basic Programming*. Sinclair Research, Cambridge

Woods, T. (1983). *Assembly Language Assembled – For the Sinclair ZX81.* Macmillan, London

Zaks, R. (1978). *Programming the Z80.* Sybex, Berkeley, California

Further Reading

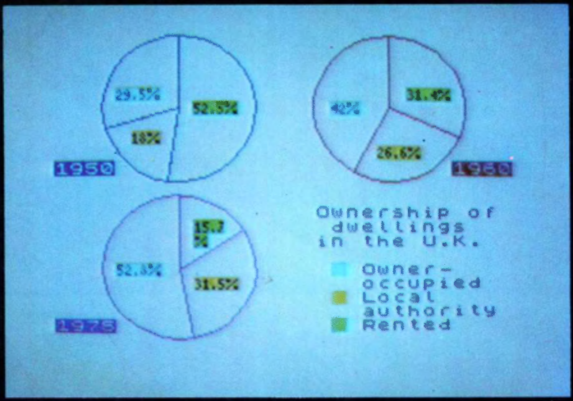
Read any periodical, magazine or journal relevant to computer graphics, such as SIGGRAPH, CADCAM, CAD journal (and there are many, many more), and the more advanced graphics textbooks (for example, Newman and Sproull, 1973), as well as the general computer newspapers and monthly magazines, such as *Personal Computer World, Practical Computing, Interface*, etc. It does not matter if you do not immediately understand the more advanced articles: it is important to appreciate the flavour, excitement and achievement of the subject. Obtain the promotional and advertising literature of the major graphics companies (Tektronix, Imlac, A.E.D., Sigma, Hewlett-Packard, D.E.C., etc.), and get as much information as possible about graphics software packages. Keep an eye on the television usage of computer graphics, whether it be in science programs, science-fiction melodramas or advertisements. Study video games and try to understand from the display how the characters are drawn and manipulated.

Index

Where to Find Routines referred to in Text

Software Cassette

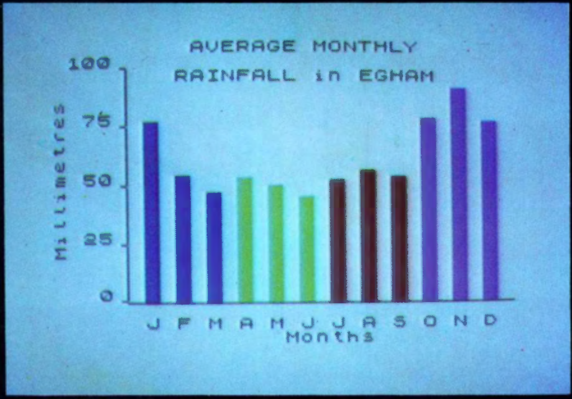
ISBN 0 333 35051 0



This book is intended for Spectrum owners who are competent BASIC programmers, but who are complete beginners in computer graphics. It contains the elementary ideas and basic information about pixel and 2-D graphics that need to be understood before the more involved concepts of character and 3-D graphics can be mastered.

A software cassette containing the programs in the book is available. See inside the back of the book for details.

Ian O. Angell and **Brian J. Jones** are both in the Department of Computer Science, Royal Holloway College, University of London.


AVERAGE MONTHLY
RAINFALL in EGHAM
Millimetres
J F M A M J J A S O N D
Months